University of Amsterdam
Faculty of Science

Master Thesis Software Engineering

Comparison of the SIG Maintainability Model and the Maintainability Index

Frank R. Oppedijk

Internship host organization	Software Improvement Group
Internship supervisor	dr. ir. J. Visser
Thesis supervisor	drs. H.L. Dekkers
Availability	unclassified
Date	25 July 2008

Contents

Abstract

Maintenance is an important aspect of the software product life cycle, maintenance effort being the single largest cost factor. The degree in which a system is easy or hard to maintain is not fixed but can be influenced. Therefore, it is important that maintainability can be measured, and subsequently acted upon.

Numerous models have been proposed that aim to measure maintainability. In this thesis work, two maintainability models are discussed. The first model is the well-known and partially validated Maintainability Index. The other model is the SIG Maintainability Model, proposed by the Software Improvement Group, where this thesis research was carried out.

As both models aim to indicate maintainability, one would expect the outcomes of the two maintainability models to show a high degree of positive statistical correlation. It was found that the two models have indeed a significant, positive correlation, but lower than was expected. Although both models were not specifically designed for measuring object-oriented programming languages, results suggest that both models are equally capable of dealing with object-oriented systems.

Further, it is shown that access routines that are typically found in object-oriented programs and which have been put forward as the reason why the Maintainability Index would not work for object-oriented systems, have only a limited influence.

Finally, both models' components are studied. For each of the models, the research indicates that several model components are strongly correlated. Both models may be improved by removing one or more components.

1 Introduction

1.1 Measuring maintainability

Maintenance is an important aspect of the software product life cycle, maintenance effort being the single largest cost factor. According to the Gartner Group, 60% to 80% of the average annual IT application development budget is spent on the maintenance of legacy applications. The degree in which a system is easy or hard to maintain is not fixed but can be influenced. Therefore, it is important that maintainability can be measured, and subsequently acted upon.

Maintainability is defined as the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment [IEEE 610.12, 1990]. As such, the maintainability of a software system is dependent not only on the product, but also on the external factors such as the person performing the maintenance and the supporting documentation and tools.

A software system's maintainability can be determined by measuring the maintenance process, e.g. the amount of time needed to carry out a modification. Fenton and Pfleeger [1997] call these external product attributes – attributes that are measured with respect to how the product relates to the environment. Drawbacks of this kind of measuring is that external attributes are usually difficult to measure correctly[1], and can only be measured late in the development process [Fenton and Pfleeger, 1997].

Numerous models have been proposed that aim to measure maintainability and do not have aforementioned drawbacks. These models use measures of internal software product attributes (e.g. relating to the structure of the product) that often can easily be extracted using static source code analysis. A good model contains a combination of internal attributes that yields a maintainability indication that closely approaches the external reading of maintainability.

1.2 Two maintainability models

In this thesis work, two maintainability models based on measuring internal product attributes are discussed. The first model is the well-known Maintainability Index (MI) [Oman and Hagemeister, 1994]. This model consists of a polynomial expression which is calculated from a number of software product metrics, and results in one number indicating the overall system maintainability. The MI has been validated multiple times for several procedural programming languages (C, Pascal, FORTRAN and Ada), and has frequently been used as a comparison model in later maintainability research. Its correctness for use with object-oriented (OO) programming languages has not been formally validated, though several papers [Welker *et al.*, 1997; Welker, 2001] argue that the MI is indeed usable for OO systems (although perhaps in a modified form, to compensate for the shorter method length of access routines ['getters and setters'] typically found in OO software systems). At least one study has used the MI as a maintainability indicator for OO systems [Misra, 2005].

The other model is the SIG Maintainability Model (SMM) [Heitlager *et al.*, 2007], proposed by the Software Improvement Group (SIG), a consultancy firm that has specialized in assessing software quality. The SIG model possesses a hierarchical structure, in which software product metrics are aggregated to four higher-level quality characteristics – analyzability, changeability, stability and testability – which together determine system maintainability. Doing so, the SMM adheres to the ISO

[1] One of the reasons is leakage. This is the phenomenon where a person spends effort on a task without registering this time spent with the task, coloring the effort measurement.

9126 standard (see also section 2.1). The SIG model has not undergone scientific validation, but has proven its commercial value in many software system assessments carried out by SIG.

Both models will be discussed in more detail in chapter 2.

1.3 Research questions

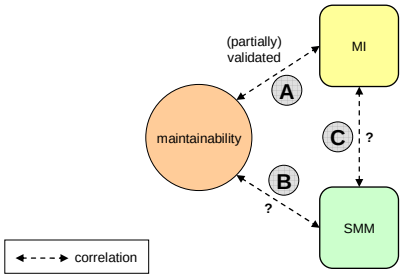

Figure 1. MI and SMM versus maintainability

Figure 1 shows both maintainability models and the relationships they have to maintainability. The relationship between maintainability and the MI (depicted by A) has been validated for a number of programming languages. As both maintainability models intend to represent the same indication of maintainability, one would expect to see a similar relationship between maintainability and the SMM (depicted by B), and therefore one would expect a high correlation of the outcomes of both maintainability models in a side-by-side comparison (depicted by C). Statistically analyzing relationship C is the subject of this thesis. Therefore, the main research question is:

RQ1.	Do the outcomes of the MI and SMM maintainability models show a high degree of positive statistical correlation, when both models are applied side-by-side on identical software systems?

As said before, the validation of the MI has been performed on a number of procedural programming languages. Because of this validation, the MI may be used as a standard for software systems written in these programming languages. Comparing the SMM to the validated MI for these programming languages acts as a validation of the SMM (for these programming languages), and thus is a special case. Therefore, the comparison for systems written in programming languages for which the MI has been validated is considered separately.

As the MI has been argued to also be applicable for software systems written in OO programming languages, this situation is also considered separately.

This leads to the following two underlying research questions:

RQ1.1	Do the outcomes of the MI and SMM maintainability models show a high degree of positive statistical correlation, when both models are applied side-by-side on identical software systems *written in languages for which the MI has been validated*?
RQ1.2	Do the outcomes of the MI and SMM maintainability models show a high degree of positive statistical correlation, when both models are applied side-by-side on identical software systems *written in OO programming languages?*

Section 1.2 mentions papers by Welker [Welker *et al*., 1997; Welker, 2001], stating that it may be the case that the small size of access routines of OO software systems influence the value of the MI. This thesis investigates this influence by answering the following research question:

RQ2.	How much influence do the access routines of OO software systems have on the value of the MI?

It is known from earlier research that many of the internal product attributes that are used in software quality models are strongly correlated. A well-known example is the strong correlation between unit size and cyclomatic complexity than has been confirmed in at least 10 studies [Shepperd and Ince, 1994]. Both the MI and the SMM models use multiple internal product attributes as model components (including unit size and cyclomatic complexity). In a good model, these model components are orthogonal and correlations are low. This leads to the last research question:

RQ3.	How strong are the correlations of the MI and SMM internal model components?

1.4 Structure of this thesis

The remainder of this document is composed as follows:

In chapter 2, a brief background about software quality models and software metrics classification is given, after which both maintainability models will be discussed in detail.

Chapter 3 discusses the research method. The research consists of five phases: exploratory phase, preparation phase, data extraction phase, analysis phase, and conclusion phase. This chapter discusses these five phases in detail.

In chapter 4, the results for the comparison between the two maintainability models are presented. Also, the results of a test for the influence of access routines on MI values are presented. Finally, results for the correlations between the internal components of both models are shown.

Chapter 5 interprets these results and presents the answers to the research questions in the form of conclusions and discussion. The very final section discusses possible future work.

2 Background

This chapter gives a brief background of software quality models and software metrics classification. Further, the MI and SMM maintainability models are discussed in detail.

2.1 Software quality models

Software maintainability is a part of the total quality of software. A well-known early software quality model, commonly called the FCM (Factor-Criteria-Metric) model, was established by McCall *et al.* [1977]. This model possesses a hierarchical structure, consisting of high-level *quality factors* that are composed of lower-level *quality criteria*, which are measured with *metrics* (see Figure 2).

The reason that software quality models are often hierarchical is because the factors that one would like to measure are too abstract to be directly measured [Fenton and Pfleeger, 1997].

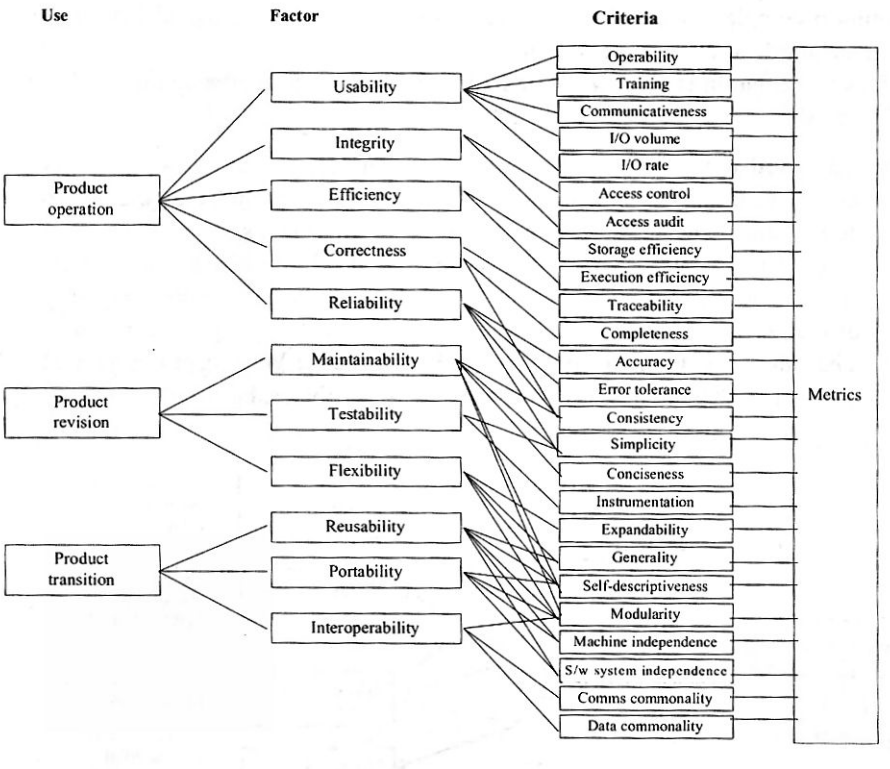

Figure 2. McCall software quality model[2]

In 1992, a derivation of the McCall model was adopted as ISO standard 9126. In 2001, an extended version was published [ISO 9126-1, 2001]. This standard defines software quality as "the totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs." Software quality is decomposed into six characteristics, see Figure 3. ISO 9126 claims to be comprehensive, meaning that any possible aspect of software quality can be described in terms of the six ISO 9126 characteristics. Each of the six characteristics can be subdivided into a total of 20 sub-characteristics.

A critique on the ISO 9126 standard is that levels below that of the six characteristics are not defined within the standard (even the 20 sub-characteristics are only defined in an annex to the standard,

[2] Reproduced from [Fenton and Pfleeger, 1997]

giving 'examples of possible definitions'), meaning that different parties with different views of software quality can select different definitions.

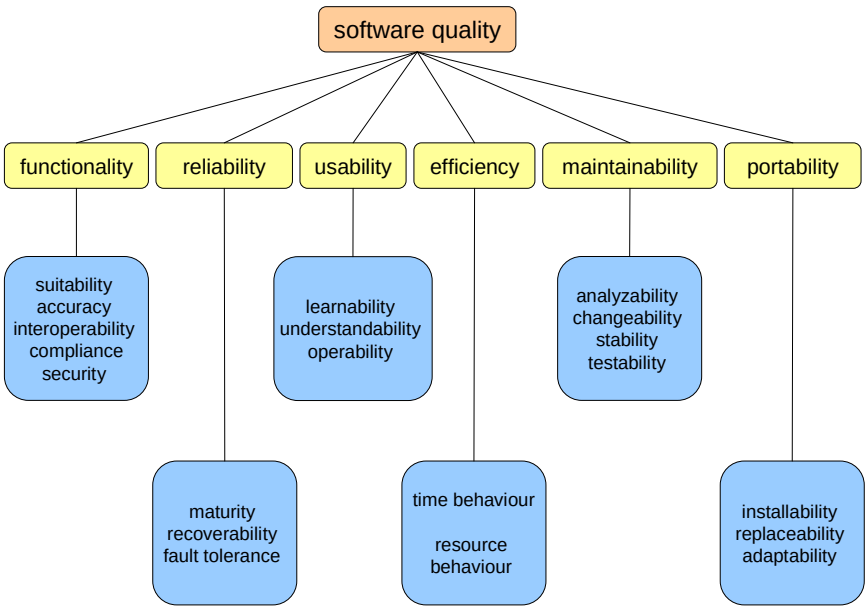

Figure 3. ISO 9126 software quality model

2.2 Classifying software metrics

In software engineering, a software *project* is carried out to produce some required software *product* via some specific software *process*. These are three important factors in software development. Software metrics can be classified in these three categories: project metrics, product metrics, and process metrics [Kan, 2003].

Project metrics describe the project characteristics and execution. Examples include project cost and developer productivity. A well-known cost/effort estimation model using project metrics is the COCOMO model by Boehm [1981]. This model consists of a number of mathematical equations that were calibrated on a database of previous project data.

Product metrics describe the characteristics of the product, such as size, complexity, and performance. Both maintainability models described in this thesis use product metrics.

Process metrics describe the software development and maintenance process. Examples are the effectiveness of defect removal, and the response time of the fix process. An example of a process model is the SEI Capability Maturity Model (CMM). In this model, a number of 'key process areas' (KPAs) have been defined which are essential for a high quality software process. A software process is ranked 1 (lowest) to 5 (highest) on these KPAs, leading to a measure of the ability to produce quality software reliably.

Fenton and Pfleeger [1997] use a slightly different classification, combining projects and processes into the processes category, and recognizing a separate category *resources*, which are entities that are required by a process activity, such as personnel and software tools.

Within each of three categories of software development, one can distinguish between internal and external attributes. Fenton and Pfleeger [1997] describe these as follows:

- *Internal attributes* are those that can be measured purely in terms of the product, process or resource itself, e.g. system size or unit complexity.
- *External attributes* are those that can be measured only with respect to how the product, process or resource relates to it environment, e.g. product usability.

The quality factor that is the subject of this thesis, maintainability, is an *external* product attribute, as the maintainability of a product is dependent on external factors such as the persons performing the maintenance and the supporting documentation and tools [Fenton and Pfleeger, 1997].

The metrics used by the two maintainability models that will be discussed in the next two sections, are *internal* product attributes. The models are built on the assumption that internal product attributes can act as predictors for external product attributes.

2.3 Maintainability Index

2.3.1 Composition of the MI

The Maintainability Index was constructed by Oman and Hagemeister at the University of Idaho, following a desire for a maintainability model consisting of a number of easily computed metrics, in order to be able to quickly and easily predict software maintainability [Oman and Hagemeister, 1994]. It consists of a polynomial expression and results in one number indicating the overall system maintainability. The MI has been calibrated based on correlation with subjective evaluations by software maintainers.

The MI exists in two variants, which only differ in the last component [SEI, 2002][3]:

$$MI3 = 171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln(aveLOC) \quad (1)$$

or

$$MI4 = 171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln(aveLOC) + 50 * sin\ (sqrt(2.4 * perCM)) \quad (2)$$

in which

aveV = average Halstead Volume V per module
aveV(g') = average extended cyclomatic complexity per module
aveLOC = average count of line of code per module
perCM = average percent of lines of comments per module

(These underlying metrics are discussed in more detail in the next section.)

The components are calculated at the module level, and then averaged. The word 'module' used here means the smallest unit of functionality. Depending on the programming language, this is a function, procedure, method, subroutine or section.

Expression (2), which includes a term for the percentage comments, should only be used if the comments are valid, instead of e.g. chunks of program code that have been commented out. Otherwise, expression (1) should be used [SEI, 2002].

[3] The original definition by Coleman *et al.* [1994] is slightly different, the last term of the four-component expression being defined as $50 * sin\ (sqrt(2.46 * perCM))$. See also section 2.3.1.1.

A higher MI value indicates better maintainability. More specifically, the following cutoff points have been identified [Oman and Hagemeister, 1994; Welker *et al*., 1997]:

Maintainability	*Expression (1)*	*Expression (2)*
High	MI >= 50	MI >= 85
Moderate	-	65 <= MI < 85
Low	MI < 50	MI < 65

Table 1. MI cutoff values

The MI has frequently been used as a comparison model in other maintainability research. A recent example is that of Samoladas *et al*. [2004].

Besides using MI at the system level, as described above, another application that is sometimes seen, calculates the MI at the module level for all modules. With this approach, one can identify the modules with the lowest MI value, which are thought to be the modules with the greatest necessity to be improved.

2.3.1.1 MI components

Below, the four components of the MI are discussed in more detail.

Lines of code

Program or module size is often measured using the Lines of Code (LOC) metric. It probably was the very first software metric, being used as the basis for measuring programming productivity and effort since the late 1960s [Fenton and Neil, 2000]. LOC still is a very popular metric, not in the least because of its simplicity.

There exist many definitions of what constitutes one LOC. Some definitions include comment lines while others don't. Some definitions count physical lines, while others count logical lines (lines with source instructions terminated by a logical delimiter such as a semicolon). According to Jones [1992], the difference between these various definitions can be as large as 500%, depending on the programming language. The MI calculation calls for counting physical lines of code [Welker *et al*., 1997].

Halstead Volume

Halstead Volume is one metric of the family of metrics known as Software Science, designed by Maurice Halstead [Halstead, 1977]. All Halstead metrics are based on four scalar numbers derived directly from a program's source code:

n1 = the number of distinct operators
n2 = the number of distinct operands
N1 = the total number of operators
N2 = the total number of operands

From these numbers, Halstead first calculated vocabulary (n) and length (N):

$n = n1 + n2$
$N = N1 + N2$

Finally, program volume (V) is calculated as:

$V = N * {}^{2}log(n)$

Halstead considered this metric to be representative for the size of any implementation of an algorithm, and also considered it to be a count of the number of mental comparisons required to generate a program. Fenton [1994] states that "the length, the vocabulary and volume of a program are considered as reflecting different views of program size." Fenton criticizes interpretation of the Halstead Volume as a measure of program complexity, such as done by the Carnegie Mellon Software Engineering Institute [SEI, 1997].

Another point of critique is that in his book, Halstead [1977] does not unambiguously define what should be considered an operator, and what should be considered an operand, but only provides a small example after which he concludes that the description of operators and operands is intuitively obvious and requires no further explanation. Al-Qutaish and Abran [2005] finely point out "intuition is insufficient to obtain accurate, repeatable and reproducible measurement results." The obscurity in Halstead's description has led to many different interpretations, especially for programming languages (such as XML) that didn't exist at the time of Halstead's writing, and which contain constructs incomparable to the example in Halstead's book.

Further criticism is that Halstead length, which is used to construct Halstead volume, overestimates program length for small programs and underestimated it for large programs [Beser, 1982; Davies and Tan, 1987].

Despite all the criticism, the Halstead metrics remain in much use, also in recent studies [Menzies *et al.*, 2002] and [Kiricenko and Ormandjieva, 2005].

Cyclomatic complexity

Cyclomatic complexity is software metric that was developed by Thomas McCabe. It is used to measure program complexity [McCabe, 1976]. It measures the number of linearly independent paths through a program's source code. Cyclomatic complexity can be calculated in two ways. In the first, a module or program is regarded as a mathematical graph. Then, the following formula is calculated:

$$V(g) = e - n + p$$

in which

e = the number of edges in a graph,
n = the number of nodes in a graph, and
p = the number of connected components

The alternative way is to count the number of decision points (if-statements, while-statements, etc) in a module or program, and increase this by one. Both methods yield the same result. The following code fragment consists of one decision point, thus yielding a cyclomatic complexity of 2.

```
IF (A=0)                                                    (3)
  THEN ...
  ELSE ...
```

A problem with this definition is whether to count the condition in compound conditional statements as one or as multiple nodes. In order words, whether the code fragment

```
IF (A=0 AND B=1)                                    (4)
  THEN ...
  ELSE ...
```

would have a cyclomatic complexity of either 2 or 3 for this fragment of code. The original McCabe would count it as V(g)=2.

Glenford Myers reasoned that compound statements such as listed in (4) as above did increase complexity, so a higher cyclomatic complexity would be appropriate. However, he also felt that the above code fragment was less complex than example (5) below, which has a cyclomatic complexity of 3:

```
IF (A=0)                                            (5)
  THEN IF (AND B=1)
     THEN ...
    ELSE ...
  ELSE ...
```

Myers proposed an extension to the cyclomatic complexity, in which compound conditional statements using AND and OR operators would receive a cyclomatic complexity *range*. In code fragment (4), the extended cyclomatic complexity would thus be 2:3 [Myers, 1977]. The extended cyclomatic complexity is denoted as V(g').

Present-day software metrics tooling often implements a subtly different implementation of V(g'), counting each component in compound conditional statements separately. In other words, the extended cyclomatic complexity of (4) would be 3.

The MI uses the extended cyclomatic complexity.

Percentage of lines of comments

The last component of the MI is the 'percentage of comments'. It is used in an expression of the form

$$50 * \sin(\sqrt{2.4 * perCM}) \qquad (6)$$

The name perCM can be somewhat confusing. The authors meant this parameter to have values ranging from 0 to 1, as can be deduced from [Pearse and Oman, 1995], in which the authors refer to perCM as the 'proportion of comments' and consider a value of 0.49 to be very high. Using this interpretation results in a monotonously increasing value for (6), see Figure 4, left. This relationship means that the higher the proportion of comments in the source code, the better, but that the beneficial effect of comments decreases as the comments proportion is higher.

Interpreting perCM as a percentage (ranging 0..100) instead, as some authors do (e.g. [Liso, 2001]), would lead to an influence that is alternating between positive and negative, see Figure 4, right. This kind of influence cannot be backed by any software engineering knowledge (except maybe for the peak at around 25% percent, which is commonly accepted as a normal value for the amount of comments [Liso, 2001]).

The factor 2.4 in (6) is mentioned as 2.46 in Coleman *et al.* [1994]. The latter causes (6) to approach the maximum value of 50 even closer, but the difference is practically insignificant.

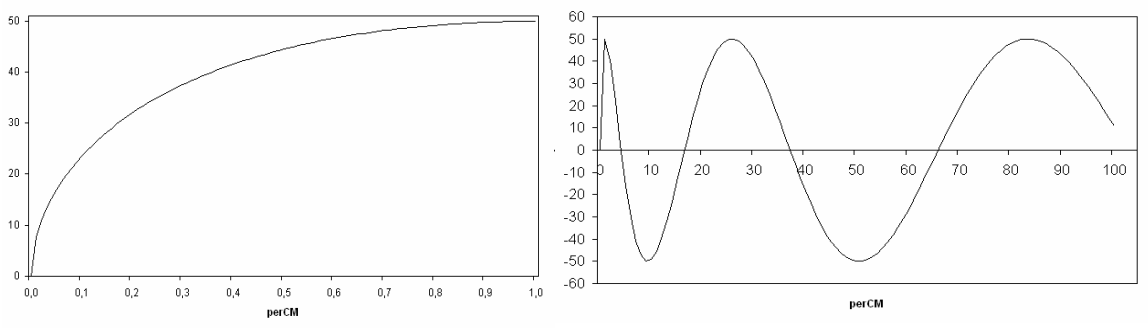

Figure 4. Percentage comments - left uses proportion 0..1; right uses percentage 0..100

2.3.2 Validation

The MI has been validated multiple times for several procedural programming languages: C, Pascal, FORTRAN and Ada, with the emphasis on the C language [Ash *et al.*, 1994; Coleman *et al.*, 1994; Oman and Hagemeister, 1994; Coleman *et al.*, 1995; Oman, 1995; Pearse and Oman, 1995]. In all these cases, the validations used subjective evaluations by maintainers of industrial software systems which were found to closely match the values predicted by the MI [Welker *et al.*, 1997]. The Software Engineering Institute agrees that the breadth of the validations performed support the claim that "the MI generally fits industrial-sized software systems" [SEI, 2002].

Some specific results of the validations mentioned above are:

- Oman and Hagemeister [1994] found a Pearson r of 0.83 (statistically significant at α=0.05) and a Spearman ϱ of 0.83 (*not* statistically significant at α=0.05[4]) on a previous version of the MI model (which in a later study turned out to not be significantly different from the definitive MI model [Welker *et al.*, 1997]). This validation used human assessments to compare the MI values against.
- Oman [1995] found a Spearman ϱ of 0.81 between measured component quality and the quality assessed by software practitioners.

The suitability of the MI for use with object-oriented (OO) programming languages has not been formally validated, though several papers argue that the MI is indeed usable for OO systems. One of these papers is by Welker *et al*. [1997], in which the authors describe a case-study in which the MI was used on a C++ system. The results show that "the MI, despite its traditional roots, might also be used to quantify maintainability in object-oriented code even if the fit is somewhat less than perfect." They continue to say that, just as procedural software systems, "object-oriented systems are primarily comprised of operators and operands, lines of code, lines of comments, and have a number of paths through a module or system as do more traditionally designed systems. Furthermore, the software maintenance practitioner still is interested in and requires the means to measure code density, size, comments and execution logic. Although object-oriented design reflects multiple levels of abstraction within the design, implementation in languages such as C++ continues to possess numerous parallels with traditional approaches. Therefore, existing MI metrics may provide a starting point."

Four years later, Welker published another article about the MI-OO subject [Welker, 2001]. He states that "MI is still great for identifying overly complex (and therefore difficult to maintain) modules in object-oriented systems."

[4] The fact that the Spearman test was not statistically significant while the Pearson test was significant, was possibly caused by the Spearman test being a nonparametric, thus less powerful statistical test (see also section 3.1.2.3).

Welker [2001] adds an interesting note on the MI values for OO systems: "It appears [...] that object-oriented systems by nature have a fairly high MI due to the typical smaller module size. Naturally, smaller modules contain less operators and operands, less executable paths, and less lines of comments and code; therefore, the MI tends to be higher. It is the author's opinion that even so, the MI is still applicable for object-oriented systems, but that maybe the maintainability classification thresholds should be raised when interpreting MI's from object-oriented systems." [Welker, 2001]

At least one study has used the MI as a maintainability indicator for OO systems [Misra, 2005].

2.3.3 Strengths and weaknesses

Oman and Hagemeister [1994] argue that a strong point of the MI is that it is easily calculated from a number of well-known metrics that are derived through static code analysis. Pearse and Oman [1995] mention another advantage, that the MI's *single value* is "useful in tracking the effects of maintenance changes on different versions of the code over time, including inter-module comparisons of complexity and comparing pre- and post-change software quality." Further, "the single index is less volatile than any of the individual metrics from which it is constructed; that is, fluctuations in one metric dimension do not inordinately change the MI, making it more stable" [Pearse and Oman, 1995].

However, the fact that the MI is a single value is also one of the model's disadvantages. As Pearse and Oman [1995] put it: "by looking at a single value you miss the detailed information provided by the raw metrics which permit you to understand the nature of the change(s) that took place. For a given source code maintenance task the value of MI changes when one or more of the polynomial's four factors change, but you can't tell just from looking at MI which of the four raw measurements is causing that change." In other words: one cannot do root-cause analysis.

As was explained earlier, the coefficients in the model were determined by calibrating the model against existing software systems. SEI [2002] advises "to test the coefficients for proper fit with each major system to which the MI is applied." In other words: the model coefficients can be system-dependent.

Further, there is criticism on several of the metrics underlying the MI. The most important criticism is about the Halstead Volume metric, for the lack of an unambiguous definition of how operands and operators should be counted. The ambiguity in the Halstead Volume calculation may lead to different MI values, depending on which definition for the Halstead Volume metric is used.

Finally, there is criticism on the way the MI is built up from components with differing scales of measurement (see section 3.1.2.3 for more information on scales of measurement), and the scale of the MI is not self-evident. Adding and subtracting the MI components required these to be on at least the interval scale. However, as Briand *et al.* [1995a] state, "often it is very difficult to determine the scale type of a measure. For example, what is the scale type of cyclomatic complexity? Can we assume that the distances on the cyclomatic complexity scale are preserved across all of the scale? This is difficult to say and the answer can only be based on intuition." Furthermore, Zuse [2005] proves that the scale type of the Halstead Volume depends on the size of the programs, changing from ratio scale (for small programs), to ordinal scale (for middle size programs), back to ratio scale (for large programs). Fenton and Pfleeger [1997] show that "the scale type for an indirect measure will generally be no stronger than the weakest of the scale types of [its components]". In conclusion: it could well be that the MI is not valid on the interval scale. This is not a problem in itself, but it reduces the range of statistical tests that may be used on MI data.

2.3.3.1 SIG criticism on the MI

SIG have expressed additional criticism on the use of MI in [Kuipers and Visser, 2007]. First, the authors argue that the average cyclomatic complexity is "a fundamentally flawed number. Particularly for systems built using OO technology, the complexity per module will follow a power law distribution. Hence, the average complexity will invariably be low (e.g. because all setters and getters of a Java system will have a complexity of 1), whereas anecdotal evidence suggests that the maintenance problems will occur in the few outliers that have exceptionally high complexity." Although the authors may have a valid point here, their argument is primarily aimed at OO systems, whilst the MI was not created for OO languages.

Further, in [Heitlager *et al.*, 2007], SIG argues that "counting the number of lines of comment, in general, has no relation with maintainability whatsoever. More often than not, comment is simply code that has been commented out, and even if it is natural language text it sometimes refers to earlier versions of the code." Again, the authors may have a point here. But commented-out code can be recognized as such by static code analysis tools and can thus be ignored, leaving only the argument that a comment "sometimes refers to earlier versions of the code." Subsequently, the authors argue that it may be that a large amount of comments have been added to the code "precisely because it is more complex, hence more difficult to maintain." Indeed, a localized high amount of comments may be indicative for complex code, but on the other hand, that liberal addition of comments may have made the combination of code and comments very maintainable. Thus, it may still make sense to use comments percentage as an indicator that has a positive effect on maintainability.

Finally, [Heitlager *et al.*, 2007] express that "there is no logical argument why the MI formula contains the particular constants, variables, and symbols that is does. The formula just 'happens' to be a good fit to a given data set. As a result the formula is hard to understand and explain."

2.4 SIG Maintainability Model

2.4.1 Composition of the SMM

The SIG Maintainability Model has evolved over a couple of years and has been first publicly proposed in [Heitlager *et al.*, 2007]. The model has a hierarchical structure and maps on the ISO 9126 maintainability characteristic and its four sub-characteristics. An example is given in Figure 5 for the changeability sub-characteristic:

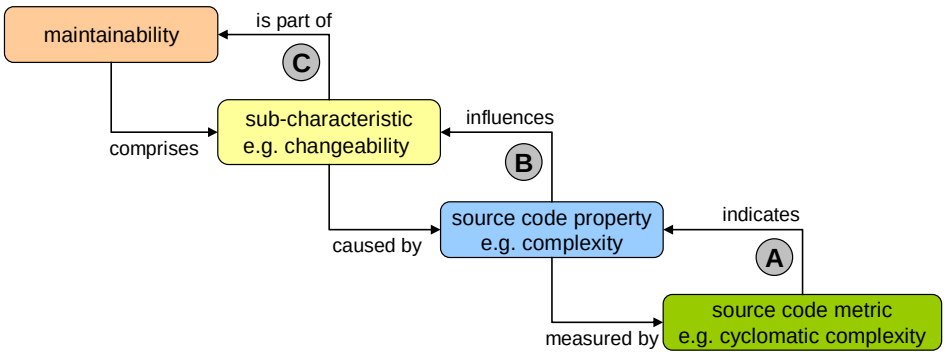

Figure 5. The SIG maintainability model

The mapping between the various levels, indicated by the A, B and C in Figure 5 is as follows.

Mapping A

The mapping between source code metrics and source code properties, as indicated by the A in Figure 5 is not linear, but consists of a rating, which is performed as follows (the example is for cyclomatic complexity, other mappings are similar):

- First, the cyclomatic complexity is calculated for each unit (being the smallest unit of functionality, i.e. a subroutine, function or method);
- Then each unit is categorized in one of the four categories of Table 2, listing the SEI categorization for cyclomatic complexity, with a weight equaling the LOC of the unit;

CC	*Risk evaluation*
1-10	simple, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk
>50	not testable, very high risk

Table 2. Cyclomatic complexity categories

- Finally, the weighed complexity percentages lead to a rating, via Table 3. This table shows the maximum allowed percentage of LOC per category to obtain a certain rating. For example, a system which has 13% of its LOC with cyclomatic complexity of 11-20, 7% of its LOC with cyclomatic complexity of 21-50, and 2% of its LOC with cyclomatic complexity over 50, will be rated with a - (the CC>50 being the most restrictive aspect in this example).

CC 11-20	*CC 21-50*	*CC >50*	*rating*
25%	0%	0%	++
30%	5%	0%	+
40%	10%	0%	o
50%	15%	5%	-
-	-	-	--

Table 3. Complexity rating

Each source code property is measured by one metric, as indicated in Table 4.

source code property	*source code metric*
volume	system LOC (converted to man-years in order to obtain language-independency)
complexity	cyclomatic complexity per unit
duplication	percentage duplication over 6 lines long
unit size	LOC per unit

Table 4. Metrics for source code properties

Mapping B

The mapping between source code properties and ISO 9126 sub-characteristics, as indicated by the B in Figure 5 is performed by calculating the simple average of all source code properties that are deemed applicable for a sub-characteristic (shown in Table 5), resulting in a score that can range from -- to ++ (or, on a numerical scale, from -2 to +2). For example, when a system has been rated a ++ for complexity and an o for duplication, it will receive a + for changeability

volume	*complexity*	*duplication*	*unit size*	*sub-characteristic*
x		x	x	analyzability
	x	x		changeability
				stability
	x		x	testability

Table 5. Mapping source code properties to ISO 9126 sub-characteristics

Note: In [Heitlager *et al.*, 2007], a fifth source code property *unit testing* is included in the model. However, in the fully automated system assessments that SIG currently performs, the four-component model shown in Table 5 is used, as unit test coverage data is not provided by the automated system assessments. This *unit testing* property is the only source code property that maps to the stability sub-characteristic.

Note: Also, for system risk assessments (SRAs) including expert judgments, a separate model consists which has five extra properties: high-level architecture, modularization, separation of concerns, exception handling, and development process.

Mapping C
The mapping between the four ISO 9126 sub-characteristics and maintainability is performed by calculating the simple average of all four sub-characteristics, resulting in a overall score that can range from -- to ++ (or, on a numerical scale, from -2 to +2). Besides the overall maintainability score, SIG also always presents the sub-characteristics ratings, as they allow root-cause analysis.

2.4.2 Validations performed

The SIG model has not undergone scientific validation. Development of the model has been driven by the results of commercial software system assessments carried out by SIG: "In the course of dozens of software assessment projects performed on business critical industrial software systems, this model has been tested and refined." [Heitlager *et al.*, 2007]

Moreover, the current SIG model is still preliminary. As [Heitlager *et al.*, 2007] explains: "adjustments and refinements are made to the model on a case by case basis. Nonetheless, the practical value of the model has already been demonstrated in our practise [sic.], and we expect further improvements of the model to only bring an increased degree of detail and precision."

2.4.3 Strengths and weaknesses

A strong point of the SMM is that the hierarchical composition of the model enables root-cause analysis. Further, the model uses metrics that are clearly defined and easily calculated (with the exception of unit test coverage, which is hard to determine using *static* source code analysis). Third, because of the categorization (mapping A), the SMM is sensitive to small changes in the system, but not so sensitive that its indication will be 'off the scale' when a large change has occurred in the system. Finally, the way the ratings are presented (on a scale ranging from -- to ++) makes them easily interpretable for management.

A weak point of the SMM is that it is as yet not validated; thus it is unknown to SMM users how good it indicates maintainability.

Further, a consequence of the SMM model still being adjusted and refined, is that SMM assessments that have been performed using an earlier version of the model cannot be simply compared to more recent assessments.

Finally, according to measurement theory [Schneidewind, 1992], the averaging done in mappings B and C of the model requires the components to be on at least interval scale (see section 3.1.2.3 for more information on scales of measurement). However, the nonlinearity created by the categorization in mapping A causes the components to be valid at only the ordinal scale. Thus, the averaging performed is not allowed, and strictly speaking, invalidates the model.

3 Research method

This chapter contains aspects concerning the research method. The research consists of five phases, which are presented in Figure 6 below. This chapter discusses these five phases in detail.

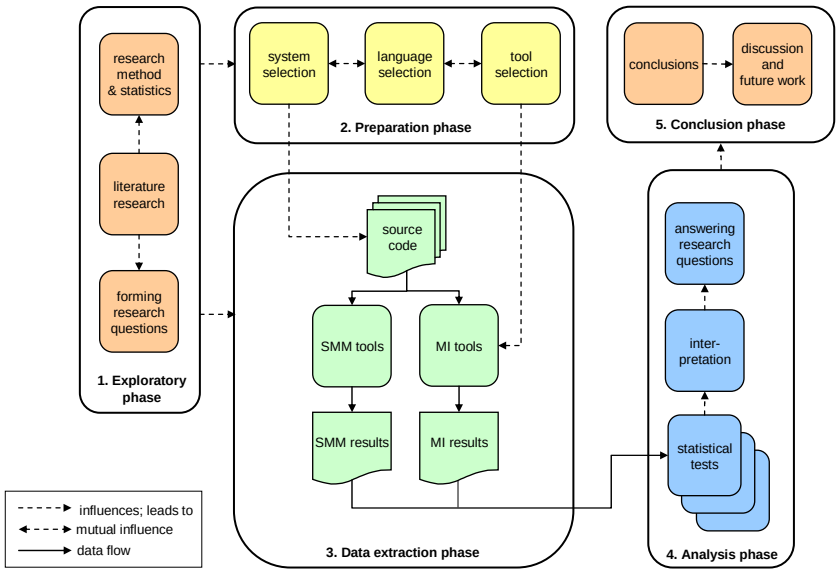

Figure 6. Research method

3.1 Exploratory phase

In the exploratory phase, literature research leads to forming research questions and to determining the appropriate research method, including the statistical approach.

3.1.1 Literature research

The literature research was carried out by performing an extensive search on both the University of Amsterdam digital library and on Google scholar, for articles regarding maintainability models in general and the MI and the SMM in particular. The resulting literature list was checked with subject-matter experts from SIG.

Statistics knowledge was obtained from the well-known textbook by Anderson *et al.* [1990]. For specific subjects, additional information was obtained with library research. Also, advice was received from Dr. M.W. van Someren of the University of Amsterdam.

3.1.2 Statistical comparison issues

Two categories of statistical tests exist: non-parametric tests and parametric tests. The former are more robust than the latter, meaning that they can be used when less can be assumed about the data. The latter are more powerful, meaning that their probability of correctly rejecting a false null hypothesis is higher. A consequence of this is that a smaller sample size can suffice for drawing a conclusion with the same degree of confidence. However, for most of the parametric tests, the data must also comply with three requirements:

1. The data must have been obtained from populations which are statistically normal;
2. The populations must have equal variances;
3. The data must be at least on the interval scale of measurement.

These three requirements will be addressed in the subsections below. After that, a subsection will discuss which statistical tests may be used in this thesis work.

3.1.2.1 Determining the normality of the populations

One cannot look at the complete population of software systems in order to assess whether it is distributed normally. Therefore, we have three alternatives. First, if the sample data set is sufficiently large, one can look at the sample data set and, if that is normally distributed, it is likely that the population is also normally distributed. Second, one may learn the distribution type from earlier research. Or last, one may reason about the probable distribution of the population.

3.1.2.2 Determining the variances of the populations

The variance of the populations can be determined by inspection of the sample data, looking at the dispersion or standard deviation of the data. The F-test can also be used to test the hypothesis that the samples have been drawn from populations with the equal variance [Anderson *et al.*, 1990].

3.1.2.3 Determining the scale of measurement

In order to choose the appropriate statistical tests, it is important to know the scales of measurement of the data to be tested. Statistical science recognizes the following four scales of measurement [Anderson *et al.*, 1990]:

- Nominal scale – the data are labels or categories, e.g. gender (man, woman);
- Ordinal scale – the data can be used to rank the observations, e.g. clothing sizes (S, M, L, etc.) or Likert scales (1=very unlikely, 2=unlikely, etc);
- Interval scale – the interval between observations can be expressed in a fixed number, e.g. temperatures (the difference between a day-time temperature of 25°C and a night-time temperature of 5°C is 20°C);
- Ratio scale – the ratio of the measurements is meaningful, e.g. prices (a car costing €60,000 is twice as expensive as a car costing €30,000).

The former two scales allow nonparametric tests to be applied, while the latter two scales also allow parametric tests to be used (provided that the populations also satisfy the other requirements).

Both the SMM and the MI ratings are at least on the ordinal scale, as from their definitions it is known that it is correct to say an SMM rating of ++ is better than a rating of +, and that an MI value of 75 is better than one of 65.

However, it is not clear if the SMM and MI are also on the interval scale: is the difference between a ++ and a + SMM rating equal to the difference between a + and a 0 rating? Is the difference between MI values 75 and 65 equal to the difference between 95 and 85? Intuitively, one would say so. Also, when looking closely at the both models, one sees that the creators assume interval-scales. Both models involve summing a number of components[5], thus indicating that they consider the model components - and thus the model – to (at least) be on the interval scale. As discussed in section 2.4.3, the SMM cannot be on the interval scale, thus the SMM should be interpreted on the ordinal scale. Section 2.3.3 concluded that it may well be that the MI is not on the interval scale either, meaning that the MI should also be interpreted on the ordinal scale.

In this thesis work, the scale of measurement will be assumed to be ordinal, meaning that the statistical tests used need to be of the non-parametric kind. The next section lists a number of applicable tests.

[5] Strictly speaking, the SMM is the average of a number of components, which of course equals the sum of the components divided by the number of components.

3.1.3 Determining the statistical tests

Comparison of the two maintainability indicators can be done in two ways. One is to determine the measure in which the two are correlated. The other is to calculate an equation defining the relationship between the two indicators. Table 6 shows the suitable statistical tests (sources: [Anderson *et al.*, 1990; Fenton and Pfleeger, 1997]). As pointed out in the previous section, this research will use the non-parametric tests.

Test type	*Non-parametric*	*Parametric*
Measure of correlation	• Spearman ϱ • Kendall τ • Chi-squared	• Pearson *r*
Equation	• Logarithmic transformation • Kendall-Theil robust line	• Linear regression

Table 6. Suitable statistical tests

As the research questions ask to identify the degree of statistical correlation, this research will be using tests meant to measure the correlation.

The Chi-squared statistical test for 'goodness of fit' needs each category to be compared to contain at least 5 items. The Kendall τ and Spearman ϱ tests do not have this restriction. As our data sets don't have any inherent categories, and making up categories in order to be able to use the Chi-squared test would be arbitrary to a certain agree, the Kendall τ and Spearman ϱ tests are more applicable.

The only assumption Kendall τ and Spearman ϱ make to the data, is that it is on at least the ordinal scale.

Fenton and Pfleeger [1997] state that the Spearman ϱ and Kendall τ tests should not be used when (many) ties are present in the data. This is an incomplete statement. There is indeed a restriction in using Spearman ϱ when ties are present in the data. The regular formula for calculating Spearman ϱ is

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}$$

in which

d_i = x_i - y_i
x_i = the rank of the i^{th} element of the first data set
y_i = the rank of the i^{th} element of the second data set
n = the number of values in each data set

This formula should not be used when ties are present. Instead, one should first rank each of the data sets (where ties receive a rank that is the average of what their ranks would otherwise be) and subsequently use the Pearson correlation coefficient on the ranks [Myers and Well, 2003]. The statistical tool that is used for this research, SPSS, uses the correct implementation and can handle ties.

According to [Hill and Lewicki, 2007], "Spearman ϱ can be thought of as the regular Pearson *r*, that is, in terms of proportion of variability accounted for, except that Spearman ϱ is computed from ranks. Kendall τ, on the other hand, represents a probability, that is, it is the difference between the probability that in the observed data the two variables are in the same order versus the probability that the two variables are in different orders."

The Kendall and Spearman tests can be performed as either one-tailed or two-tailed tests. When the direction of the association between the variables is known in advance (e.g. when one hypothesizes that there is a large and *positive* correlation between the SMM and the MI), one may use the one-tailed variant. If the direction of the association is not known in advance and either a significant positive and a significant negative association would be of interest, one should use the two-tailed variant [Anderson *et al.*, 1990]. The p-value for a two-tailed test is twice as high as for a one-tailed test.

As the Kendall τ and Spearman ϱ are equivalent in their usability, and because of the similarity of Spearman ϱ with the well-known and familiar Pearson *r*, Spearman ϱ was chosen as the statistical test to use to determine the measure of association in this thesis research. This research uses one-tailed tests if the direction of the association is known in advance, and two-tailed tests otherwise.

3.1.4 Determining which correlation value is considered good

3.1.4.1 General guidelines

The following general rule of thumb exists as to how to interpret the various values for the (rank) correlation coefficient (see Table 7).

r or ρ	*interpretation*
0.00 - 0.19	very weak, or no relationship
0.20 - 0.39	weak
0.40 - 0.59	moderate
0.60 - 0.79	strong
0.80 - 1.00	very strong

Table 7. Interpreting correlation coefficients

It must be added that interpreting correlation coefficients must also take the context into account. In social sciences, finding a correlation of 0.4 in a human-behavior related study might be considered good, while in medical research, a correlation of 0.8 may be interpreted as being unacceptably low.

Specific to this research, a more specific way of interpreting correlation coefficients is given in the next two sections.

3.1.4.2 Situation A: languages for which the MI has been validated

As explained in section 2.2, both the SMM and the MI models use internal product attributes to arrive at an indication that approaches the (external) product attribute maintainability. As Fenton and Pfleeger [1997] state, this correlation cannot be perfect, as the complexity of an external attribute can never be fully described by a combination of internal attributes.

In section 2.3.2, it was stated that MI validations have shown the Spearman correlation between the MI model and the maintainability as assessed by experts was about 0.81 to 0.83. This correlation was considered very good, and it may have been one of the factors that the MI has been applied as maintainability indicator in much academic research. In Figure 7, this correlation has been depicted as $\varrho_A \approx 0.8$.

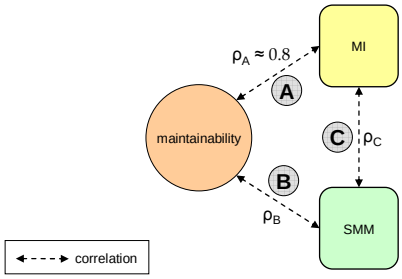

Figure 7. MI and SMM versus maintainability

This research does not measure the correlation between maintainability and the SMM, but, if the SMM model would be of comparable quality as the MI model, ϱ_B could be around a value of 0.8 as well.

The Spearman ϱ correlation coefficient indicates how much of the variability of the dependent variable is accounted for by the independent variable. Thus, ϱ_A being ≈ 0.8 means that 0.8 of the maintainability variability is accounted for by the MI model. In Figure 8, this is depicted graphically for both the MI and the SMM. The SMM-MI correlation is the area between the dashed lines.

In the extreme case that the MI and SMM models both account for maximally different fractions of the variability, the SMM-MI correlation is expected to be 1 - 0.2 - 0.2 = 0.6 (see Figure 8, top). In the other extreme case where the MI and SMM models account for identical fractions of the variability, the correlation between the two models will equal 1 (see Figure 8, bottom).

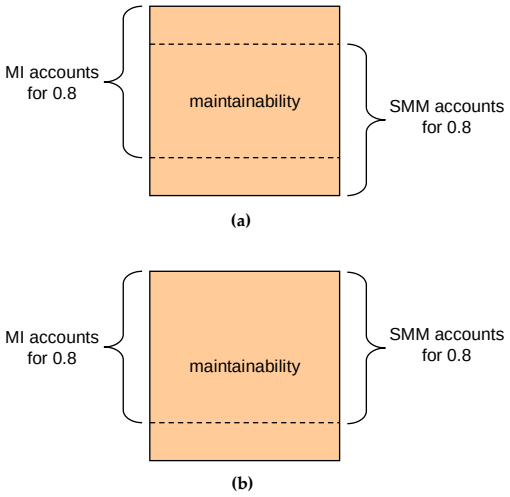

Figure 8. Variability accounted for by MI and SMM

Thus, a reasonable value to be found for the Spearman rank correlation between the MI and SMM maintainability models would be $0.6 \leq \varrho_C \leq 1$, assuming that $\varrho_B = 0.8$. More generally, one can expect ϱ_C to be at least equal to $1 - (1 - \varrho_A) - (1 - \varrho_B)$, which equals

$$\rho_C = \rho_A + \rho_B - 1 \qquad (7)$$

In order for the correlations to have sufficient statistical significance, we state that the results should be significant at the α=0.05 level. This is a common statistical practice; also, it equals the significance levels of the MI validation work.

3.1.4.3 Situation B: OO languages

The above reasoning is valid only for the case where one may assume that $\rho_A \approx 0.8$, thus only for the languages for which the MI has been validated. For OO languages, for which the MI has not been validated, there is no such guideline.

Still, as has been said in section 2.3.2, Welker *et al.* [1997] reason that, just like procedural systems, "object-oriented systems are primarily comprised of operators and operands, lines of code, lines of comments, and have a number of paths through a module or system." It is true that OO programs also have additional aspects that may have an influence on maintainability, such as abstraction and inheritance, which the MI does not measure, and this is why Welker [2001] considers using the MI for OO programs "a starting point."

An important point to consider is that both the MI and the SMM are alike in this respect: Both models consist of internal product attributes related to the structure and size of programming units, and both models lack metrics regarding typical OO aspects. Therefore, it is assumed that both models are equally good in dealing with OO systems.

Another effect OO programs have, is their smaller average unit size (because of the access routines), which influences the MI [Welker, 2001; Heitlager *et al.*, 2007]. As the SMM does not contain a component for *average* unit size, but looks for outliers (units with very large unit size), the effect of a smaller average unit size on the SMM may very well differ from the effect on the MI.

However, as the *complete* sample will suffer from the above-mentioned effect, and as we are looking at the models' rank correlations instead of correlations of their absolute values, this will not influence the rank ordering within the sample. As a consequence, the smaller average unit size of the sample does not influence the value of the SMM-MI correlation[6].

In conclusion: for systems written in OO languages, correlations that are similar to those specified in the previous section are expected to be seen.

3.2 Preparation phase

The preparation phase deals with selecting software systems, written in a selection of languages. Also, appropriate tools have to be selected. This section closes with an overview of the research data selected.

3.2.1 Software system selection

The metrics that go into both SMM and the MI model must be obtained by means of static source code analysis. Thus, in order to be able to compare the SMM and MI, source code must be available.

SIG keeps a source code repository containing the source code of many systems that have been assessed by SIG, either in a one-time Software Risk Assessment or in an ongoing Software Monitor. The source code repository was used in selecting 52 closed-source software systems.

Besides source code from the SIG repository, 21 open-source systems were included.

System selection is heavily dependent on which programming languages should be included in the comparison. This is the subject of the next section.

[6] Yet as soon as samples containing a mixture of systems with differing smaller-average-unit-sizes are studied (such as a sample containing both C and Java programs), this argument no longer holds, and we may see differing correlations.

3.2.2 Programming languages selection

In the selection process of which programming languages to include in the comparison, three factors were of influence. First, it had to be possible to calculate both the SMM and the MI for each programming language. This may sound obvious, but calculating the MI requires calculation of the Halstead Volume metric. The algorithm for calculating Halstead Volume for languages such as XML lacking a notion of operators and operators similar to that of FORTRAN, Pascal, or C, would be rather arbitrary (see also section 2.3.1.1).

Second, in order to answer research questions RQ1.1 and RQ1.2, systems written in procedural languages (for which MI has been validated), and systems written in OO languages must be selected. For statistical purposes, in order to reduce the dependency on one language for the OO programming paradigm, multiple languages must be included. Therefore, two languages per programming paradigm were chosen.

Third, and also for statistical purposes, it was desired that an ample amount of source code be available for all languages chosen. An investigation into the SIG source code repository yielded Table 8. This shows that the largest part of source code (measured in lines of code) in the repository consists of Java code, followed by C code, etc.

Rank	*Language*	*%*
1	Java	24.4%
2	C	18.6%
3	XML	10.6%
4	C++	7.6%
5	C#	7.3%
6	ABAP	6.7%
7	COBOL	4.3%
8	Informix 4GL	4.1%
9	ADABAS	4.0%
10	JSP	2.5%

Table 8. Languages in the repository

Of the languages for which the MI has been validated (C, Pascal, FORTRAN and Ada), the SIG repository contained only C systems. Therefore, only one language will be included for answering RQ1.1

For the OO programming languages for answering RQ1.2, the decision was to include Java and either C++, because these are the OO languages of which the most source code is available in the SIG source code repository.

3.2.3 MI tools selection

With the choices for the programming languages described in section 3.2.2 in mind, a tools selection phase was carried out. Ideally, the tools calculate MI directly, or alternatively, the tools should calculate unit level Halstead Volume, Cyclomatic Complexity or Extended Cyclomatic and LOC, after which the MI can be calculated fairly easily.

First, it was determined which features the tool(s) should have:

- Calculate metrics by performing static source code analysis;
- Calculate system-level MI (both 3-component and 4-component variants). Alternative is calculation of Halstead Volume, Cyclomatic Complexity or Extended Cyclomatic and LOC at the unit level;
- Command-line mode, in order to support batch processing;
- Sets of source files identified by file lists or filter expressions;
- Output written to text file, for further processing.

Second, an Internet search was done for available metrics tools. Also, papers from the literature study dealing with MI validation were studied for clues about tools used. Besides languages supported, the availability of a student license was also investigated. The outcome of this research phase is shown in Table 9.

Tool name	*Java*	*C*	*C++*	*C#*
Power Software Essential Metrics Java	x			
Power Software Essential Metrics C/C++		x	x	
Testwell CMT Java	x			
Testwell CMT++		x	x	
McCabe IQ	x	x	x	x
Virtual Machinery JHawk	x			
Pro Et Con Java FGM	x			
Abraxas CodeCheck		x	x	

Table 9. MI tools

Based on this information and the availability of student licenses, the products by Power Software, Testwell and McCabe were obtained and installed.

A preliminary test revealed that for the languages C and C++ the McCabe IQ suite requires either preprocessed source files to be available in the code base, or a preprocessor for the source platform to be available. As these preprocessed files were not available, and neither was the source platform, the McCabe IQ tool suite could not be used for C and C++.

The Essential Metrics documentation lacked a specification of whether the 3-component or the 4-component was calculated by the tool set. Neither could the tool manufacturer provide this information. Because of this lack of information, it was decided not to use the Essential Metrics tool set.

As a result of the above findings, it was decided to perform all data extraction with the Testwell CMT Java and CMT++ tool sets.

3.2.3.1 Assessment of MI calculation precision

As has been pointed out in section 2.3.3, the value of the MI may be influenced by the Halstead Volume definition used. In order to assess the magnitude of this effect in the MI tools selected, a comparative test was performed using three MI tools available, on the source code of the 15 OSS systems.

It was reasoned that differences in the exact MI values given by the various tools were acceptable, but that the rank order of the results should be highly consistent. The thought behind this reasoning is that the statistical tests used during the analysis phase will be non-parametric tests that need ordinal

scale but not interval scale on the measurement results. Hence, it is important that rank order is not influenced, but the exact values of the measurement results (thus, their intervals) are not important.

Table 10 shows the outcomes of this test. As can be seen, the Spearman rank correlation between the McCabe and CMT tools is very strong, although not perfect. The correlation between the EM tool and the CMT tool is somewhat lower, although still very strong. The McCabe and EM correlation is lowest (and is not significant). Please note that the McCabe tool was only used for the Java systems, reducing the sample size to 5. This may explain the lower significance for the correlations with the McCabe tool.

	McCabe - CMT			*CMT-EM*[7]			*McCabe - EM*		
MI variant	*corr.*	*sign.*	*n*	*corr.*	*sign.*	*n*	*corr.*	*sign.*	*n*
MI3	0.900	0.019	5	0.862	0.000	15	0.500	0.196	5
MI4	0.895	0.020	5	0.839	0.000	15	0.154	0.402	5

Table 10. Spearman rank correlation coefficients, on 15 OSS systems

The error may be caused by different Halstead Volume implementations of the various tools. Also, other differences, such as parse errors or different interpretations of the other metrics (such as LOC, see also section 2.3.1.1) may be of influence.

The influence of the tools on the outcome is considered acceptably small.

3.2.4 Sample data for the comparison

At the end of the preparation phase, 73 systems, written in C, C++ and Java were available for the research. Source data was available from 52 closed-source systems that had previously been issued to SIG in order for a software risk assessment to be carried out, as well as 21 open-source systems.

Table 11 shows descriptive statistics of these software systems. Besides containing the information for the three separate programming languages, it also contains information for the OO subset (containing the C++ and the Java systems) and for all systems. As can be seen, the data contains systems of various system sizes. This is also visible in Figure 9, which shows histograms of system size on linear and logarithmic scales. The linear histogram shows that the majority of the systems are small. On a logarithmic scale, the distribution of systems turns out to be bi-modal, with the peaks at ln(LOC)≈11 (LOC≈60k) and ln(LOC)≈12.5 (LOC≈270k).

The number of C and OO systems should be sufficient for obtaining statistically significant results. No other noticeable aspects are visible from the data.

language	*#systems*	*min size (KLOC)*	*max size (KLOC)*	*mean size (KLOC)*	*std dev size (KLOC)*	*min #units*	*max #units*	*mean #units*	*std dev #units*
C	19	9.8	4581	717	1393	136	116752	15235	31404
C++	11	13.0	558	184	169	859	21279	7024	6471
Java	43	1.1	1031	150	209	137	85942	14527	18582
OO	54	1.1	1031	157	200	137	85942	12998	17054
all	73	1.1	4581	303	759	136	116752	13581	21486

Table 11. Descriptive statistics of software systems researched

[7] Please note that the EM tools only give one MI value, and that we do not know whether this is the MI3 or the MI4. The correlation values suggest it is the MI3.

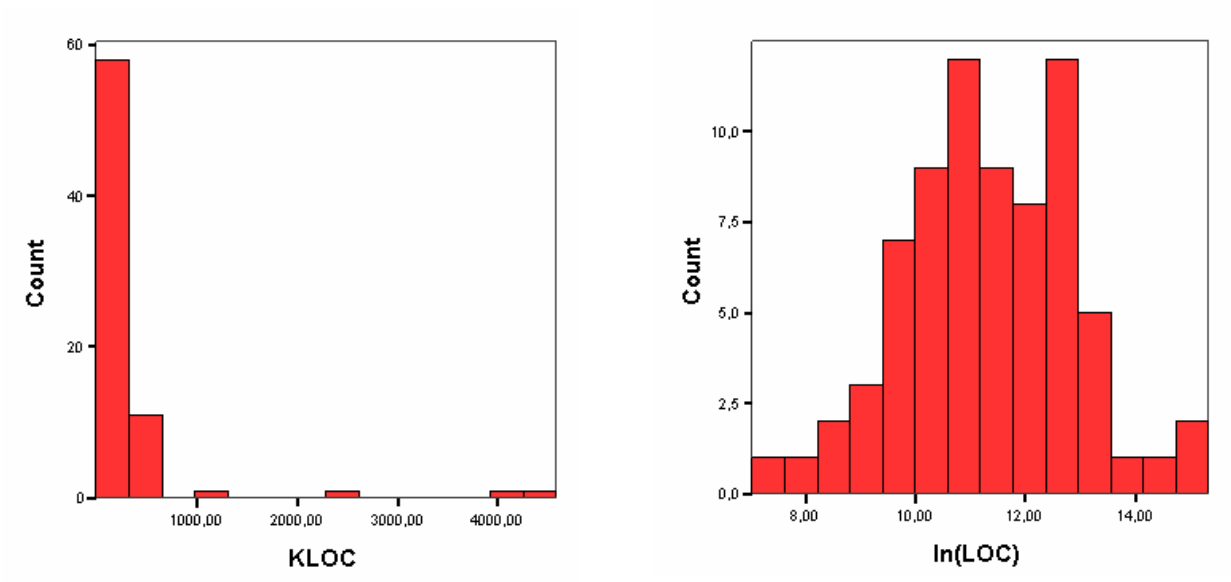

Figure 9. Size distribution of software systems researched

3.3 Data extraction phase

The data extraction phase consists of passing the source code of all selected software systems through the tools calculating the maintainability indication for both the maintainability models.

3.3.1 Determining the influence of access routines

In order to determine the influence of access routines on the MI value for RQ2, the MI3, MI4 and the MI components were calculated with the data from the access routines removed. As the way the MI3 and MI4 values are calculated differs from the MI values mentioned elsewhere in this thesis, this is explained below.

Whilst the MI values mentioned elsewhere were calculated directly by the MI tools, the MI values of for determining the influence of the access routines were calculated by taking the unit-level output from the MI tools for the MI components aveV, aveV(g'), aveLOC, and perCM and subsequently calculating the MI3 and MI4. This alternative approach enabled filtering out the data for the access routines without needing to modify all source files.

In order to eliminate any differences that may have been introduced by this alternative calculation method, it was used for both the part with access routines and the part without access routines.

3.3.2 Studying the models' components

In order to be able to study the correlations between the models' components for RQ3, the values for the components of both models had to be available for all systems.

The SMM component values are readily available, as they are reported with each run of the SMM tools. The MI component values were determined by taking the unit-level component values from the MI tools and subsequently calculating the system averages from this.

3.4 Analysis phase

In the analysis phase, the obtained results for both maintainability models undergo a number of statistical tests, after which the results are interpreted and the research questions are answered.

Research questions RQ1 (and the underlying questions RQ1.1 and RQ1.2) will be answered by running the data extraction phase on multiple software systems for which the MI has been validated, as well as on multiple OO software systems, and subsequently comparing the SMM and MI results.

Research question RQ2 will be answered by running the data extraction phase once again, on OO software systems which had all access routines removed.

Finally, research question RQ3 will be answered by collecting the models' components values for both models during the data extraction phase and comparing these.

3.5 Conclusion phase

In the conclusion phase, results obtained in the analysis phase will be used to draw conclusions. This phase will also comprise a carefully balanced discussion, and report on aspects that can be covered in future work.

4 Results

In this chapter, the results from the data extraction and the statistical tests are presented. Spearman correlation coefficients will be given for both comparisons. Also, the results of a test for the influence of access routines on MI values are presented. Further, Spearman correlation coefficients will be given for the multicollinearity between the internal components of both models. Finally, the similarities between the MI3 model and a number of simplifications to it are shown.

4.1 Raw data

The results from the data extraction phase for all systems used in the comparison are listed in full in Table 22 in Appendix A.

Table 12 below shows the descriptive statistics of these results. It displays SMM, MI3 and MI4, as well as SMM and MI components. The data are separated on language, shown for the OO subset, and for all systems together. For each of these, the minimum, maximum and median values in the sample data are shown.

As can be seen, the SMM results for the data set run from -2 to roughly +1, while the SMM scale runs from -2 ('--') to +2 ('++'). In other words, the systems in the data set do not cover the complete range of the SMM scale.

In Figure 10, box plots are shown for SMM overall maintainability, MI3 and MI4, for the same five sample groups. In each box plot, the box runs from the first to the third quartile of the sample data (also called the interquartile range), approximately covering the middle half of the observed data. The whiskers at each side of the box show the distance from the end of the box to the largest and smallest observed values that are less than 1.5 box lengths from either end of the box. Observed values between 1.5 and 3 box lengths from either end of the box are called *outliers* and are shown as an 'o'. Values over 3 box lengths away from the box are called *extremes* and are shown as an '*'.

Figure 10 shows various outliers and/or extremes. The calculations for these systems were checked but no errors could be found. Therefore, these data points are regarded as valid. The systems in question were not removed from the data set.

		SMM								*MI*					
language	*type*	*maint*	*ana*	*chg*	*tst*	*dup*	*comp*	*usize*	*vol*	*MI3*	*MI4*	*aveV*	*aveV(g')*	*aveLOC*	*perCM*
C	min	-2.000	-2.000	-2.000	-2.000	-2.000	-2.000	-2.000	-2.000	49	81	565.4	3.479	21.89	0.08
	max	0.611	0.667	1.000	0.500	2.000	2.000	-1.000	2.000	89	124	3359.1	12.491	163.88	0.44
	median	-1.278	-0.667	-1.500	-2.000	-1.000	-2.000	-2.000	2.000	76	103	862.9	5.301	40.95	0.23
C++	min	-1.889	-1.667	-2.000	-2.000	-2.000	-2.000	-2.000	-1.000	58	97	361.9	2.368	16.99	0.09
	max	0.000	0.000	0.500	-0.500	1.000	1.000	-2.000	2.000	97	119	1621.4	6.304	76.89	0.36
	median	-1.056	-1.000	-1.000	-2.000	-1.000	-2.000	-2.000	0.000	85	115	636.9	3.319	26.68	0.18
Java	min	-1.889	-1.667	-2.000	-2.000	-2.000	-2.000	-2.000	-1.000	81	109	133.4	1.288	7.37	0.03
	max	0.889	1.000	1.500	1.000	2.000	2.000	0.000	2.000	113	150	713.6	3.405	29.56	0.42
	median	-0.611	0.000	-0.500	-1.500	0.000	-1.000	-2.000	2.000	100	136	243.0	1.985	14.05	0.28
OO	min	-1.889	-1.667	-2.000	-2.000	-2.000	-2.000	-2.000	-1.000	58	97	133.4	1.288	7.37	0.03
	max	0.889	1.000	1.500	1.000	2.000	2.000	0.000	2.000	113	150	1621.4	6.304	76.89	0.42
	median	-0.750	-0.333	-0.500	-1.500	0.000	-1.000	-2.000	1.000	98	132	277.3	2.228	14.90	0.27
all	min	-2.000	-2.000	-2.000	-2.000	-2.000	-2.000	-2.000	-2.000	49	81	133.4	1.288	7.37	0.03
	max	0.889	1.000	1.500	1.000	2.000	2.000	0.000	2.000	113	150	3359.1	12.491	163.88	0.44
	median	-0.944	-0.333	-1.000	-1.500	-1.000	-1.000	-2.000	1.000	93	124	395.2	2.776	18.21	0.24

Table 12. Descriptive statistics of results

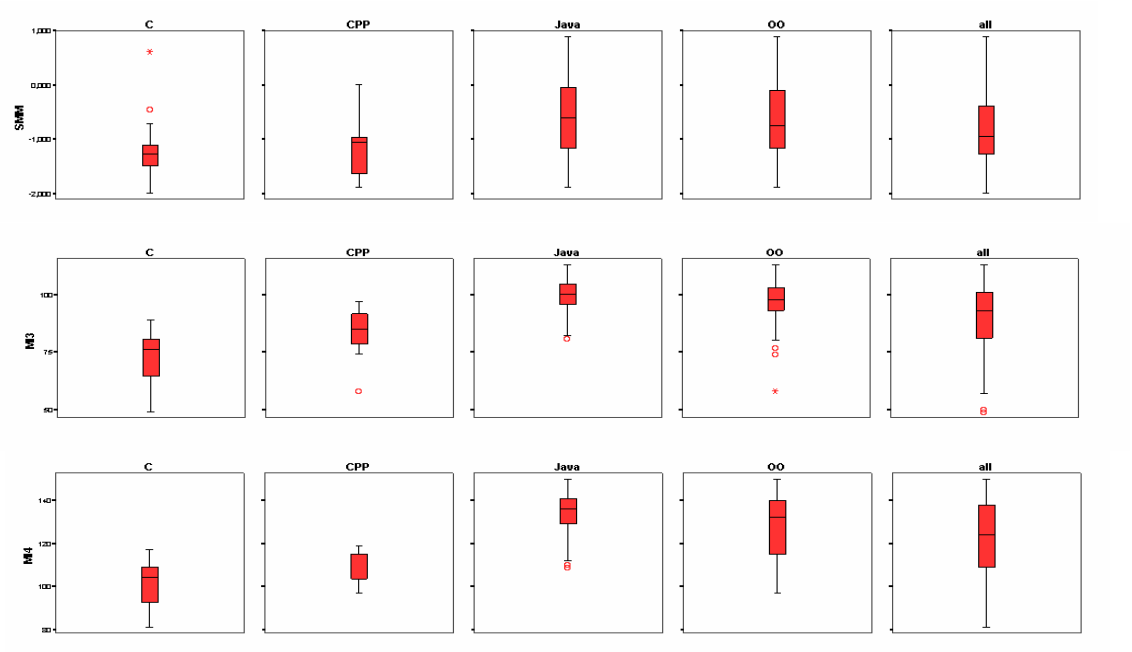

Figure 10. SMM, MI3 and MI4 box plots

4.2 Rank correlation of SMM and MI

The scatter plots of Figure 11 show SMM being plotted against MI3 and MI4. Clearly visible is that SMM tends to be higher for higher MI3 and MI4 values, indicating a possible positive correlation.

Interesting to see is also, that the results seem to be clustered by language. In general the C systems score worst, C++ systems generally score higher, and Java systems perform best. This effect is most pronounced on the MI axes, but seems to be also visible on the SMM axis.

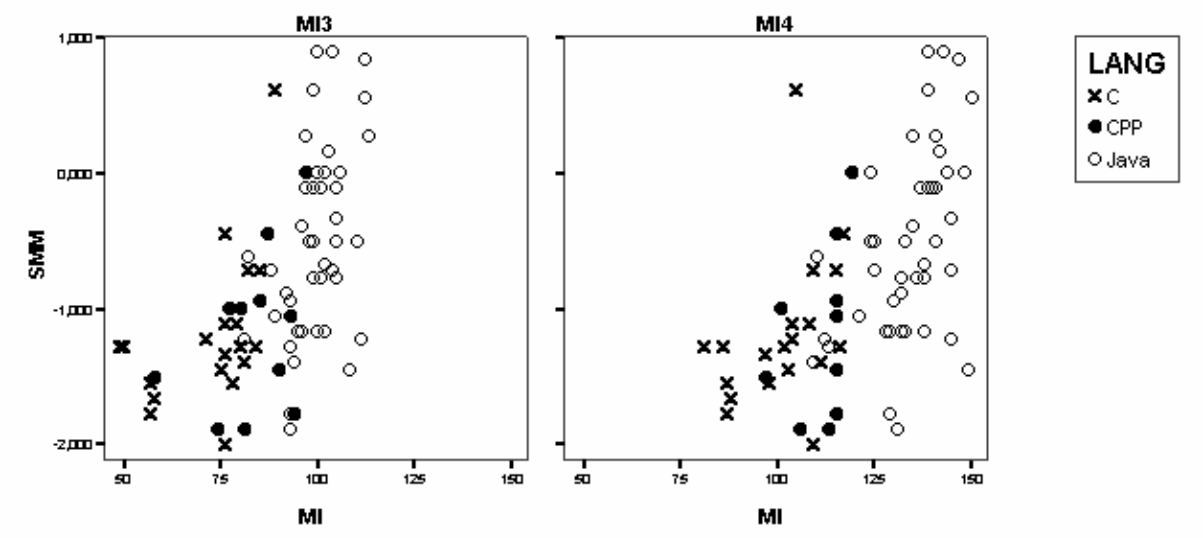

Figure 11. Scatter plots of MI3 and MI4 against SMM

As discussed in section 3.1.3, the Spearman rank correlation test is used to determine the correlation between the two maintainability models. The outcome of this test is shown in Table 13.

	SMM - MI3			SMM - MI4		
language	*corr.*	*sign.*	*n*	*corr.*	*sign.*	*n*
C	0.494	0.016	19	0.476	0.020	19
C++	0.365	0.135	11	0.423	0.098	11
Java	0.459	0.001	43	0.500	0.000	43
OO	0.554	0.000	54	0.569	0.000	54
all	0.621	0.000	73	0.617	0.000	73

Table 13. Rank correlation of SMM versus MI3 and MI4

The SMM show a significant correlation to MI3 or MI4 for the C and Java systems (significance being better than α=0.05), but not for the C++ systems.

Interesting to see, is that the correlation for all systems together is higher than the correlation of any of its subsets, for both SMM-MI3 and SMM-MI4.

Ranking the 73 systems on both SMM, MI3 and MI4, and subsequently sorting the systems on MI3 or MI4 yields the line diagram of Figure 12. If the SMM-MI correlation would have been perfect (+1.0), a monotonously rising line would have been visible, with the difference between SMM rank numbers for systems adjacent on the MI3 or MI4 axis being equal to 1. As it is, Figure 12 shows lines that jump up and down. The average absolute differences in SMM rankings for systems that are adjacent on the MI3 or MI4 axis is 16.0 (SMM-MI3) or 16.2 (SMM-MI4).

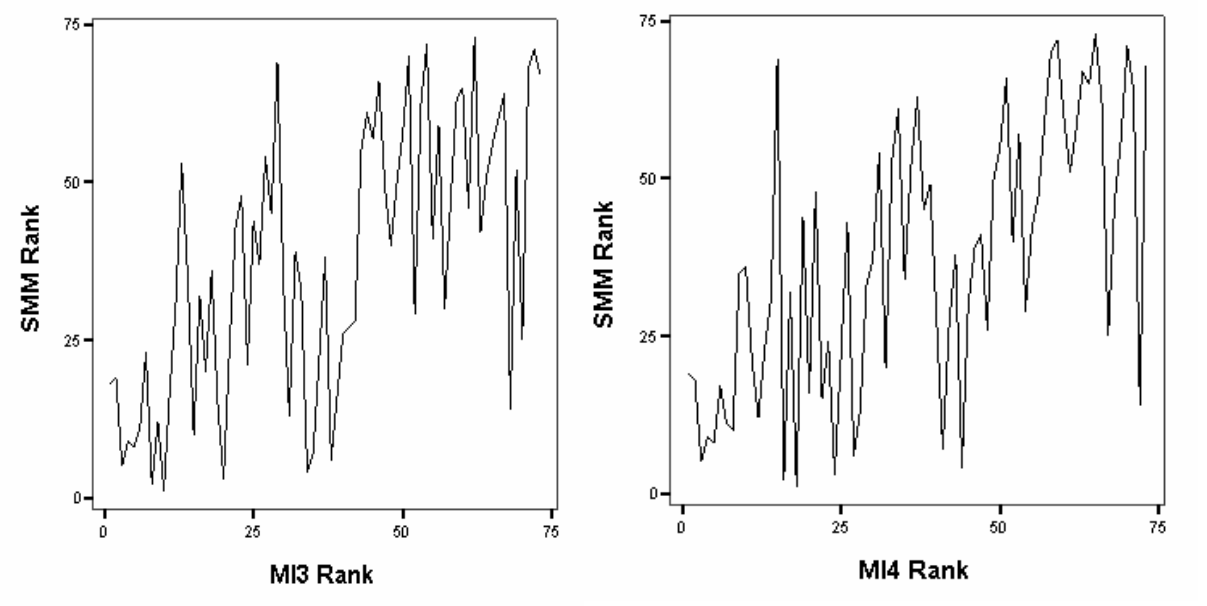

Figure 12. MI3 and MI4 ranking against SMM ranking

4.3 Influence of access routines on MI values

In order to calculate the influence of access routines on the MI value, the MI3, MI4 and the MI components were recalculated for all 73 software systems that were available, with the data concerning the access routines removed. The percentages of routines removed per language are shown in Table 14.

The exercise of removing the access routines was carried out not only for the systems in the OO languages C++ and Java, but also for the C systems, as it may have been the case that the (primarily

OO) practice of providing access routines for data attributes had also been applied in some of the C systems.

language	*#units (with access routines)*	*# units (without access routines)*	*diff (%)*
C	224016	222448	-1%
C++	55209	44632	-19%
Java	510918	221552	-57%
OO	566127	266184	-53%

Table 14. Number of access routines per language

The effect of removing the data concerning the access routines are shown in Table 15. This table shows the data with access routines on the left, and the data without access routines on the right.

As can be seen, removing the access routines causes an increase in the average Halstead volume, average extended cyclomatic complexity and average unit size, while perCM remains stable, leading to a drop in MI3 and MI4 values of 3 for the C++ systems and 8 for the Java systems[8].

	with access routines						*without access routines*							
language	*aveV*	*aveV(g')*	*aveLOC*	*perCM*	*MI3*	*MI4*	*aveV*	*aveV(g')*	*aveLOC*	*perCM*	*MI3*	*MI4*	*diff MI3*	*diff MI4*
C	1396.6	6.457	61.64	0.23	70	103	1405.3	6.470	61.95	0.23	70	103	0	0
C++	757.7	3.802	32.40	0.19	82	112	829.4	4.116	36.58	0.19	79	109	-3	-3
Java	271.9	2.128	13.65	0.27	100	135	458.4	2.816	19.66	0.27	92	127	-8	-8
OO	370.8	2.469	17.47	0.25	96	130	534.0	3.081	23.10	0.25	89	123	-7	-7

Table 15. Influence of access routines on MI values

Removing the access routines when calculating the MI, also changes the rank order of the MI3 and MI4 scores, as Table 16 shows. If removing the access routines had no influence on the rank ordering, correlations between the MI with versus without access routines had been 1,000. As it is, the correlations are somewhat lower than that, indicating a (limited) change in rank ordering (significances are one-tailed).

	MI3 (without access routines)			*MI4 (without access routines)*		
	corr.	*sign.*	*n*	*corr.*	*sign.*	*n*
MI3 with access routines	0.922	0.000	73	-	-	-
MI4 with access routines	-	-	-	0.942	0.000	73

Table 16. Correlation between MI values with and without access routines

4.4 Multicollinearity between internal model components

In Table 17 and Table 18, the Spearman rank correlations between the SMM components have been listed. The significances shown are the two-tailed variant (as we do not know in advance whether any correlation will be positive or negative).

Table 17 shows the correlation between the maintainability sub-characteristics analyzability, changeability and testability. The correlations are all significant at the 0.01 level. The correlations changeability-analyzability and changeability-testability are strong, indicating that these SMM components are not orthogonal entities.

[8] Strictly speaking, subtracting MI values is only allowed if the MI is valid on at least the interval scale, which isn't beyond dispute (see section 2.3.3).

Table 18 shows the correlation between the source code properties duplication, complexity, unit size and volume. Here, weak correlations are visible. All correlations are significant at the 0.05 level, and are lower than between the sub-characteristics.

	SMM chg			*SMM tst*		
	corr.	***sign.***	***n***	***corr.***	***sign.***	***n***
SMM ana	0.822	0.000	73	0.485	0.000	73
SMM chg	-	-	-	0.755	0.000	73

Table 17. Rank correlation of SMM sub-characteristics

	SMM comp			*SMM usize*			*SMM vol*		
	corr.	***sign.***	***n***	***corr.***	***sign.***	***n***	***corr.***	***sign.***	***n***
SMM dup	0.245	0.037	73	0.296	0.011	73	0.389	0.001	73
SMM comp	-	-	-	0.628	0.000	73	0.286	0.014	73
SMM usize	-	-	-	-	-	-	0.302	0.009	73

Table 18. Rank correlation of SMM source code properties

Table 19 shows the Spearman rank correlation between the MI components (again, significances are two-tailed). As can be seen, very strong correlations exist between aveV, aveV(g') and aveLOC (all significant at α=0.01), indicating that the model components are not orthogonal. The perCM component has a smaller and less significant correlation to the other model components, suggesting this is more of an orthogonal component than the other three.

	aveV(g')			*aveLOC*			*perCM*		
	corr.	***sign.***	***n***	***corr.***	***sign.***	***n***	***corr.***	***sign.***	***n***
aveV	0.953	0.000	73	0.952	0.000	73	-0.301	0.010	73
aveV(g')	-	-	-	0.913	0.000	73	-0.283	0.015	73
aveLOC	-	-	-	-	-	-	-0.136	0.252	73

Table 19. Rank correlation of MI components

Finally, Table 20 shows the correlation between MI3 and MI4 (one-tailed). As can be seen, the correlations are strong to very strong, with high significance.

	MI3 - MI4		
	corr.	***sign.***	***n***
C	0.700	0.000	19
C++	0.922	0.000	11
Java	0.759	0.000	43
OO	0.852	0.000	54
all	0.912	0.000	73

Table 20. MI3-MI4 comparison

4.5 Fitting a simpler MI model

As Table 19 in the previous section showed, the MI components aveV, aveV(g') and aveLOC have a very strong correlation. Table 21 shows the results of replacing the MI3 model by a simpler, one-component model MI'. The table shows three one-component models, each based on one of the components of the original MI3 model. These were obtained by fitting a linear regression line of the same form as the component in the original MI3 model.

Besides the rank correlation coefficients of the new MI' versus the original MI3, the table also shows the coefficient of determination r^2 of the linear regression line.

MI' expression	*MI3- MI' rank correlation corr.*	*sign.*	*n*	r^2
197,46 - 17,717 * ln(aveV)	0.983	0.000	73	0.966
110,07 - 5,7036 * aveV(g')	0.937	0.000	73	0.877
151,48 - 20,242 * ln(aveLOC)	0.998	0.000	73	0.995

Table 21. Simpler MI models

5 Conclusions, discussion and future work

This chapter interprets the results of this research and presents the answers to the research questions in the form of conclusions and discussion. The very final section discusses possible future work.

5.1 Conclusions and discussion

5.1.1 Research question RQ1, RQ1.1 and RQ1.2

This thesis research has looked at the correlations between the SMM and the MI maintainability models. As both models attempt to show the same indication of maintainability, a high agreement in the outcomes of both models was expected. This was voiced in research question RQ1 (see sidebar). As this question differs for software systems written in languages for which the MI has been validated and software systems written in OO programming languages, the question was split into RQ1.1 and RQ1.2.

RQ1 - Do the outcomes of the MI and SMM maintainability models show a high degree of positive statistical correlation, when both models are applied side-by-side on identical software systems?

RQ1.1 - Do the outcomes of the MI and SMM maintainability models show a high degree of positive statistical correlation, when both models are applied side-by-side on identical software systems *written in languages for which the MI has been validated?*

RQ1.2 - Do the outcomes of the MI and SMM maintainability models show a high degree of positive statistical correlation, when both models are applied side-by-side on identical software systems *written in OO programming languages?*

The answers to RQ1.1 and RQ1.2 will first be discussed in detail, after which RQ1 is answered.

5.1.1.1 Research question RQ1.1

It is concluded that the MI and SMM maintainability models show a moderate, significant degree of positive correlation, when both models are applied side-by-side on identical software systems written in the programming language C.

The SMM-MI correlation for C programs is 0.494 (SMM versus MI3) or 0.476 (SMM versus MI4) (see Table 13). Both are significant correlations (at α=0.05). This correlation was somewhat lower than the minimum expected value of 0.6 that was expressed in section 3.1.4.2.

Using formula *(7)* from section 3.1.4.2, one sees that this lower value may indicate that the relation between the SMM and maintainability ϱ_B (see Figure 7, here reproduced as Figure 13) is less than the expected 0.8. However, it may also be that, for the sample of 19 C programs, the relation between the MI and maintainability (ϱ_A) is smaller than the ≈0.8 that have been found in the MI validations. To test this, expert assessments would be necessary.

Additional discussion about the meaning of this lower correlation is in section 5.1.1.3.

Furthermore, it is important to realize the limitation of the conclusion. In this research, only one procedural programming language was included. Therefore, the answer to RQ1.1 is valid for C only, and cannot be generalized to the other three languages for which the MI has previously been validated (the - now fairly uncommon - languages Pascal, FORTRAN, and Ada[9]).

[9] A survey on job advertisements that mention programming language skills do not mention Pascal, FORTRAN or Ada in the top-25 most wanted language skills. C is at number 2, Java at 4, C# at 6, and C++ at 9 [Enticknap, 2007].

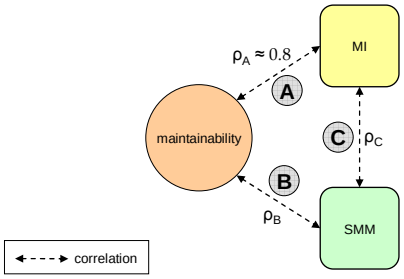

Figure 13. MI and SMM versus maintainability

5.1.1.2 Research question RQ1.2

It is concluded that the outcomes of the MI and the SMM maintainability models shows a moderate, significant degree of positive correlation, when the models are applied on the OO systems in this research.

Let's first discuss the limitations of the positive correlation with the OO sample. The C++ systems form a minority in the OO sample (11 C++ systems versus 43 Java systems). Therefore, the research results from the OO sample largely reflect the Java results.

Further, the decision was made to limit the OO languages choices to two (Java and C++), which is a limited subset of all OO languages available, although containing two of the three most popular OO languages (the third being C#) [Enticknap, 2007]. The research findings therefore cannot be directly extrapolated to C# or other languages, although the results may be valid for these languages to a certain extent.

For Java, the research shows significant SMM-MI correlations of 0.459 (SMM-MI3) and 0.500 (SMM-MI4). Again, these correlations are lower than expected. Possible reasons for this discrepancy equal those given for the C systems, i.e. that either ϱ_A or ϱ_B is lower than assumed. More discussion on the meaning of this in section 5.1.1.3.

Interesting to see, is that the SMM-MI correlations for the Java systems are similar to those for the C systems. This seems to confirm the assumption (made in section 3.1.4.3) that the fact that both models lack specific OO metrics makes them equally capable of dealing with OO systems.

It must be noted that the results for the C++ language systems have insufficient significance, which could well have been caused by the small sample size of 11 systems. Briand *et al.* [1995a] state that for a population correlation of 0.6, a sample size of about 20 is needed for the Spearman test (with a one-tailed significance of 0.05 and a power of 80%). For lower population correlations, the sample size must be even larger. As the sample correlations for the C++ systems, 0.365 (SMM versus MI3) and 0.423 (SMM versus MI4) do not suggest to be population correlation to be larger than 0.6, this indicates that 11 may indeed be too small a sample size. In order to give a decisive answer, a larger sample size would be needed. However, this was not available in this research.

Another way to circumvent the insufficient significance would have been to use more parametrical statistical tests, which usually are more powerful. As we haven't been able to determine that the MI and SMM are valid on the interval scale, the common statistical knowledge is that we cannot use parametric tests. However, Briand *et al*. [1995b] state that "with care and after thorough reflection" one could still use certain parametric tests in these cases. "Many common parametric techniques (e.g., product moment correlation, t-test) are robust to non-linear distortions of interval scales as long as they are not 'too extreme', e.g., exponential. If this is the case, these statistics will have a tendency to

be conservative and underestimate existing relationships or differences" [Briand *et al*., 1995b]. Studies such as Oman and Hagemeister [1994] have indeed used parametric statistical tests on the MI, without proof of the MI being valid on the interval scale.

Finally, the SMM-MI correlations for the OO programs as a whole (combining the C++ and Java samples), as well as the SMM-MI correlations for all languages together (combining the C, C++ and Java samples), show higher correlations than each of their subsets. This has likely been caused by the fact that the data are rather clustered by language, as can be seen in the scatter plots of Figure 11. This clustering leads to high between-subset rank correlations, which in turn raises the correlations for the supersets.

5.1.1.3 Research question RQ1

As was pointed out in the previous two sections, the SMM-MI correlations for both the language for which the MI has undergone previous validation, as well as for the OO languages included in this research are moderately high, positive, and significant, but lower than expected. It has been stated in the previous sections that this may have been caused by ϱ_A and/or ϱ_B being lower than assumed. In other words: either the MI model or the SMM model, or both, are worse indicators of maintainability than previously assumed. As we have not measured ϱ_A and ϱ_B directly, we cannot know whether this is the case. Measuring ϱ_A and ϱ_B would call for a different research approach, using expert assessments in order to get to know a system's maintainability.

Another consideration as to the lower than expected correlations is the following. Both models have a different approach to determining maintainability. While the MI model looks at system-wide averages, the SMM uses a hybrid approach, in which extremes (such as one unit with a cyclomatic complexity over 50) can be detected, while also looking at the big picture (by using categorization (see section 2.4.1, mapping A). This makes the SMM more sensitive to small changes, which can cause the SMM value to change while the MI value stays unaffected (this is visualized by the large vertical jumps in Figure 12). Obviously, this can influence the rank ordering and thus lower the rank correlation. Also, the SMM moves in a non-linear way because of the categorization in the model. A jump in the SMM value may still cause a change in the rank ordering and subsequently a lower rank correlation, even if both the MI and the SMM react to a change[10].

Still, the (significant) correlations found in this research are in the range that is considered indicating a moderate relationship, using the rule of thumb given in section 3.1.4.1. This indicates that indeed, both models do measure the same phenomenon (and the $\varrho_A \approx 0.8$ found in the validation described in section 2.3.2 suggests that this phenomenon is maintainability). The fact that both models are implemented in a different way may be the cause of the lower than expected correlation. This research does not decide on how good each of the models is at indicating maintainability, or which one is the better model. But again, this was not the intention of this research.

The fact that the correlations for the languages included in this research are quite similar suggests that the SMM-MI correlation is fairly language-independent, implying that both SMM and MI are equal in indicating maintainability for either procedural programming languages or OO programming languages. The language-independence is confirmed by looking at the model components: the MI can be calculated for each language that allows the calculation of its components

[10] For the sake of completeness it must be said that the MI also moves in a slightly non-linear way because the MI values are usually (as by the tool used in this research) rounded to whole numbers. This effect, however, is quite small for the MI (the non-linear 'jump' by the rounding is only 2 to 3% [MI4 and MI3, respectively] of the total MI range seen in this research, while on average this is 19% for the SMM).

(languages such as XML are notorious exceptions - see section 2.3.1.1); the SMM was even designed to be language-independent [Heitlager *et al.*, 2007].

Having said this, it may be the case that the SMM-MI correlation measured in this research is also valid for programming languages outside the scope of this research. However, this needs to be investigated further.

5.1.2 Research question RQ2

RQ2 - How much influence do the access routines of OO software systems have on the value of the MI?

Welker [2001] remarks on the MI values for OO systems that "It appears [...] that object-oriented systems by nature have a fairly high MI due to the typical smaller module size." Research question RQ2 deals with determining the extent of this influence.

The conclusion is that, while the OO systems researched indeed have a higher average MI than the procedural systems, the influence of access routines on the height of the average MI of OO systems is limited. The research has shown that access routines account for one-quarter to one-third of the higher MI that OO systems have, compared with procedural systems.

It was shown that the amount of access routines varies considerably per programming language (C++: 19%, Java: 53%) and the effect of this on the MI varies per language as well (C++: 3 MI points, Java: 8 MI points) (see Table 14 and Table 15). Still, the difference in average MI of C++ and Java systems when compared with C systems is much higher: the median C++ MI3 is 9 points higher than the median C MI3, while the median Java MI3 is 24 points higher than the median C MI3. For the MI4 these differences are 12 and 33 points, respectively (data from Table 12). From these data, it can be calculated that the influence of access routines on the MI values of OO systems is limited to 24% to 33%[11].

The remaining part of the difference in MI values is caused by OO programs having smaller overall unit sizes (not limited to just the access routines) as well as lower unit complexity, when compared to procedural programs. This is shown in Table 15: the average unit size and average complexity of OO systems without access routines are over twice as small as those of the C systems. Apparently, OO programmers create smaller and less complex units than their procedural counterparts.

It must be noted, however, that both the SMM and the MI measure unit complexity only *within* units. Creating a system consisting of many very small units could introduce possible complexity-increasing aspects such as high coupling and excessive inheritance. These aspects are measure by OO-specific metrics such as *fan-out* and *depth of inheritance tree*, which are not part of either the SMM or MI. Therefore, such influences cannot be captured by these models.

Finally, removing the access routines when calculating the MI changes the rank order of the MI3 and MI4 scores very little. As Table 16 shows, the rank correlation between the MI version with access routines and the MI version without access routines are very strong. This means that the rank ordering on the MI scores with and without access routines are very much alike.

[11] Please note that this calculation assumes the MI to be valid on the interval scale, which isn't beyond dispute (see section 2.3.3).

5.1.3 Research question RQ3

RQ3 - How strong are the correlations of the MI and SMM internal model components?

Both maintainability models are a composition of multiple components. As part of this research, the correlations between the components of each model were studied (see Table 17 through Table 20).

A common rule of thumb is that each pair of model components should have a correlation coefficient in the range $-0.5 \leq r \leq 0.5$. When a correlation coefficient outside this range is encountered, one of the two variables of the pair could be removed from the model [Kooiker, 1997]. Anderson *et al*. [1990] use a slightly different rule of thumb. They state that multicollinearity becomes a problem if the absolute value of the sample correlation coefficient exceeds 0.7.

Irrespective of which rule of thumb is applied, one reaches the same conclusion: that the SMM components changeability and analyzability, as well as the SMM components changeability and testability may have too high a multicollinearity. The same goes for all pairs of the MI components average Halstead Volume, average extended cyclomatic complexity and average unit size. Both models might benefit from removing one or more components. It must be noted, that SIG currently already prefers to interpret the SMM components (analyzability, changeability and testability) separately, instead of the overall SMM maintainability rating, because of the possibilities of root-cause analysis, as described in section 2.4.3.

Table 21 shows three possible simpler MI models, each containing just one of the MI3's components. As can be seen, there are very strong rank correlations between the original model and the simplified ones, and also the coefficient of determination r^2 of the linear regression line is very high. This supports the opinion expressed above that the MI3 model can be simplified[12].

As Table 19 shows, the percentage comments component, part of the MI4, is the model component with the smallest (absolute) value of multicollinearity, suggesting that this component is more of an orthogonal component than the other three. Yet, the SMM does not contain a comments component, as SIG argues that "more often than not, comment is simply code that has been commented out." While this may be true, commented-out code should be pretty well recognizable for static code analysis tools, and thus could be ignored. The SMM might benefit from the addition of such a comments component.

The MI perCM component has a *negative* correlation to the other three MI components (Table 19). However, as the sign of the perCM in the MI formula (see section 2.3.1) is opposite from the sign of the other three components, the perCM component still affects the MI value in the same direction as the other three MI components do (although less pronounced than the other three, because of its lower multicollinearity). This also explains why the MI3 and the MI4 (whose only mutual difference is the perCM component) still have such a strong multicollinearity (see Table 20).

5.2 Future work

This thesis work has studied the MI and SMM maintainability models, comparing how they perform as maintainability indicators. It has done so by looking at the level of correlation between the two models. As has been pointed out in section 3.1.4.2, a correlation anywhere in the range from 0.6 to 1.0 may indicate that the SMM performs just as good as the MI. The breadth of this range, which is

[12] It must be added that the high r^2 score in itself is not proof that the suggested simplified MI' expressions are appropriate to replace the MI3. Also, a residual analysis must be undertaken to confirm that the linear regression expression fits properly.

caused by the fact that the SMM was judged by comparing it to the MI instead of comparing it directly to maintainability, limits the power of this research.

Another limitation of the research is that it cannot tell which of the two maintainability models performs better. In order to be able to do so, one would ideally compare both models to maintainability as assessed by human experts. Unfortunately, performing human expert assessments on a large number of software systems is an arduous task.

An alternative approach may be to use an indicator that has been shown to be strongly correlated to maintainability or maintenance effort, and which can be obtained in an automated way. Li and Henry [1993] have successfully used the number of changed source code lines as an indicator for maintenance effort. As maintainability has been defined as the ease with which a software system or component can be modified, it can also be regarded as the number of change requests that can be implemented per unit of maintenance effort. Or, taking Li and Henry's findings into account, as the number of change requests divided by the number of changed lines.

The approach suggested in the previous above was not followed in this thesis research as the system change history was not available for most of the systems. However, to conclude our research, we did carry out a small study using change request and changed lines data collected by Yu [2006] on 117 versions of the Linux kernel. When both the SMM and the MI model were calculated on these Linux systems, the results indicated significant (at α=0.05) but weak correlations of 0.175 for the SMM and 0.320 for the MI4, while the MI3 did not show a significant correlation. These weak correlations may have been caused by limitations of this preliminary study.

One limitation of this study is that all change requests were regarded equal, which of course is a bold assumption. Yu [2006] concludes that a corrective maintenance task requires more effort than the other types of maintenance task. Another limitation is that each changed source code line was regarded equal. With present-day refactoring tools, changing a hundred lines of code in an automated refactoring may be very unlike to changing a hundred lines by hand. Still, using a maintainability indication that is obtained in an automated way may be a promising alternative for performing a wide-scale study into the performance of both the MI and the SMM maintainability indicators for a variety of present-day programming languages.

Bibliography

[Al-Qutaish and Abran, 2005] Al-Qutaish, R.E., and Abran, A., *An analysis of the design and definitions of Halstead's metrics*, In 15th Int. Workshop on Software Measurement, 2005, pp. 337-352.

[Anderson *et al.*, 1990] Anderson, D.R., Sweeney, D.J., and Williams, T.A. *Statistics for Business and Economics*, 9th ed. West Pub. Co., 1984.

[Ash *et al.*, 1994] Ash, D., Alderete, J., Yao, L., Oman, P., and Lowther, B., *Using software maintainability models to track code health*. In Proceedings of the international conference on software maintenance, 1994, pp. 154-160.

[Beser, 1982] Beser, N., *Foundations and experiments in software science*. In Selected Papers of the 1982 ACM SIGMETRICS Workshop on Software Metrics: Part 2, 1982, pp. 48-72.

[Boehm, 1981] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981.

[Briand *et al.* 1995a] Briand, L., El Emam, K., and Morasca, S., *Theoretical and Empirical Validation of Software Product Measures*. Technical Report number ISERN-95-03, International Software Engineering Research Network, 1995.

[Briand *et al.* 1995b] Briand, L., El Emam, K., and Morasca, S., *On the application of measurement theory in Software Engineering*. Technical Report number ISERN-95-04, International Software Engineering Research Network, 1995.

[Coleman *et al.*, 1994] Coleman, D., Ash D., Lowther, B., and Oman, P., *Using Metrics to Evaluate Software System Maintainability*. In IEEE Computer, 1994, vol. 27(8), pp. 44-49.

[Coleman *et al.*, 1995] Coleman, D., Lowther, B., and Oman, P., *The application of software maintainability models in industrial software systems*. In Journal of Systems and Software, 1995, vol. 29(1), pp. 3-16.

[Davies and Tan, 1987] Davies, G., and Tan, A., *A note on metrics of Pascal programs*. In ACM SIGPLAN Notices, 1987, vol. 22(8), pp. 39-44.

[Enticknap, 2007] Enticknap, N., *IT salary survey: finance boom drives IT job growth*. In SSL/Computer Weekly. Online version: http://www.computerweekly.com/Articles/2007/09/11/226631/sslcomputer-weekly-it-salary-survey-finance-boom-drives-it-job.htm. Visited July 2008.

[Fenton and Neil, 2000] Fenton, N.E. and Neil, M., *Software metrics: roadmap*. In Proceedings of the Conference on the Future of Software Engineering, 2000, pp. 357-370.

[Fenton and Pfleeger, 1997] Fenton, N.E. and Pfleeger, S.L., *Software Metrics. A rigorous & practical approach.* PWS Publishing Company, 1997.

[Fenton, 1994] Fenton, N., *Software Measurement: A Necessary Scientific Basis.* In IEEE Transactions on Software Engineering, 1004, vol. 20(3), pp. 199-206.

[Halstead, 1977] Halstead, M.H., *Elements of Software Science*. Elsevier, 1977.

[Heitlager *et al.*, 2007] Heitlager, I., Kuipers, T. and Visser, J., *A Practical Model for Measuring Maintainability*. In 6th International Conference on the Quality of Information and Communications Technology, 2007, pp. 30-39.

[Hill and Lewicki, 2007] Hill, T., and Lewicki, P., *STATISTICS Methods and Applications*. StatSoft, 2007. Online version: http://www.statsoft.com/textbook/stathome.html. Visited July 2008.

[IEEE 610.12, 1990] IEEE, *IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology*. Standards Coordinating Committee, 1990.

[ISO 9126-1, 2001] *ISO/IEC 9126-1:2001, Software engineering -- Product quality -- Part 1: Quality model*, 2001.

[Jones, 1992] Jones, C., *Critical Problems in Software Measurement*. Software Productivity Research, 1992.

[Kan, 2003] Kan, S. H., *Metrics and Models in Software Quality Engineering*, 2nd ed. Addison-Wesley, 2003.

[Kiricenko and Ormandjieva, 2005] Kiricenko, V., Ormandjieva, O., *Measurement of OOP size based on Halstead's Software Science*. In Proceedings of the 2nd Software European Forum, 2005, pp. 253-259.

[Kooiker, 1997] Kooiker, R., Marktonderzoek, Wolters Noordhoff, 1997

[Kuipers and Visser, 2007] Kuipers, T., and Visser, J., *Maintainability Index revisited – position paper*. In 11th European Conference on Software Maintenance and Reengineering, 2007.

[Li and Henry, 1993] Li, W., and Henry, S., *Object-Oriented Metrics that Predict Maintainability*. In Journal of Systems and Software, 1993, vol. 23(2), pp. 111-122.

[Liso, 2001] Liso, A., *Software Maintainability Metrics Model: An Improvement in the Coleman-Oman Model*. In Crosstalk, Journal of Defense Software Engineering, 2001, pp. 15-17.

[McCabe, 1976] McCabe, T.J., *A Complexity Measure*. In IEEE Transactions on Software Engineering, 1976, vol. 2(4), pp. 308-320.

[McCall *et al.*, 1977] McCall, J.A., Richards, P.K., and Walters, G.F., *Factors in Software Quality*. General Electric Co., 1977.

[Menzies *et al.*, 2002] Menzies, T., Di Stefano, J.S., Chapman, M., and McGill, K., *Metrics That Matter*. In Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop, 2002.

[Misra, 2005] Misra, S. C., *Modeling Design/Coding Factors That Drive Maintainability of Software Systems*. In Software Quality Control, 2005, vol. 13(3), pp. 297-320.

[Myers and Well, 2003] Myers, J.L. and Well, A.D., *Research Design and Statistical Analysis*, 2nd ed. Lawrence Erlbaum, 2003.

[Myers, 1977] Myers, G.J., *An extension to the cyclomatic measure of program complexity*. In SIGPLAN Notices, 1977, vol. 12(10), pp. 61-64.

[Oman and Hagemeister, 1994] Oman, P., and Hagemeister, J., *Construction and testing of polynomials predicting software maintainability*. In Journal of Systems and Software, 1994, vol. 24(3), pp. 251-266.

[Oman, 1995] Oman, P., *Applications of an automated source code maintainability index*. Technical Report number 95-08-SL, Software Engineering Test Laboratory, University of Idaho, presented at the 1995 Software Technology Conference, 1995.

[Pearse and Oman, 1995] Pearse, T. and Oman, P., *Maintainability measurements on industrial source code maintenance activities*. In Proceedings of the international Conference on Software Maintenance, 1995, pp. 295-303.

[Samoladas *et al.*, 2004] Samoladas, I., Stamelos, I., Angelis, L., and Oikonomou, A., *Open source software development should strive for even greater code maintainability*. In Communications of the ACM, 2004, vol. 47(10), pp. 83-87.

[Schneidewind, 1992] Schneidewind, N.F., *Methodology for Validating Software Metrics*. In IEEE Transactions on Software Engineering, 1992, vol. 18(5), pp. 410-422.

[SEI, 1997] VanDoren, E., *Halstead Complexity Measures*. Software Engineering Institute, Carnegie Mellon University, 1997. Online version: http://www.sei.cmu.edu/str/descriptions/halstead.html. Visited July 2008.

[SEI, 2002] VanDoren, E., *Maintainability Index Technique for Measuring Program Maintainability*. Software Engineering Institute, Carnegie Mellon University, 2002. Online version: http://www.sei.cmu.edu/activities/str/descriptions/mitmpm_body.html. Visited July 2008.

[Shepperd and Ince, 1994] Shepperd, M. and Ince, D. C., *A Critique of Three Metrics.* In the Journal of Systems and Software. Elsevier Science, 1994, vol. 26(3), pp. 197-210.

[Welker *et al.*, 1997] Welker, K.D., Oman, P.W., and Atkinson, G.G., *Development and application of an automated source code maintainability index.* In Journal of Software Maintenance, 1997, vol. 9(3), pp. 127-159.

[Welker, 2001] Welker, K.D., *The Software Maintainability Index Revisited.* In Crosstalk, Journal of Defense Software Engineering, 2001.

[Yu, 2006] Yu, L., *Indirectly predicting the maintenance effort of open-source software*. In Journal of Software Maintenance and Evolution: Research and Practice, 2006, vol. 18(5), pp. 311-332.

[Zuse, 2005] Zuse, H., *Resolving the mysteries of the Halstead Measure*. In Metrikon 2005 – Software-Messung in der Praxis, 2005. Online version: http://www.horst-zuse.homepage.t-online.de/z-halstead-final-05-1.pdf. Visited July 2008.

Appendix A - SMM and MI results

Table 22 shows the results from the data extraction of the SMM and MI on 73 software systems. The columns have the following meaning (from left to right):

- system ID
- system programming language
- SMM overall maintainability
- SMM analyzability, changeability, and testability sub-characteristics
- SMM duplication, complexity, unit size, and volume source code properties
- 3-component and 4-component MI
- average Halstead Volume, average extended cyclomatic complexity, average LOC, and average proportion of comments (all at the unit-level)

		SMM								*MI*					
#	*lang*	*maint*	*ana*	*chg*	*tst*	*dup*	*comp*	*usize*	*vol*	*MI3*	*MI4*	*aveV*	*aveV(g')*	*aveLOC*	*perCM*
1	C	-1.778	-1.333	-2.000	-2.000	-2.000	-2.000	-2.000	0.000	57	87	2704.3	9.263	115.61	0.19
2	C	-1.278	-0.333	-1.500	-2.000	-1.000	-2.000	-2.000	2.000	49	86	3359.1	12.491	163.88	0.36
3	C	-1.278	-0.333	-1.500	-2.000	-1.000	-2.000	-2.000	2.000	50	81	3307.0	10.345	132.37	0.28
4	C	-1.333	-1.000	-1.500	-1.500	-2.000	-1.000	-2.000	1.000	76	97	746.5	5.301	32.25	0.14
5	C	-1.667	-1.000	-2.000	-2.000	-2.000	-2.000	-2.000	1.000	58	88	3085.3	10.873	130.85	0.17
6	C	-1.556	-0.667	-2.000	-2.000	-2.000	-2.000	-2.000	2.000	57	87	3005.9	10.759	130.12	0.26
7	C	-2.000	-2.000	-2.000	-2.000	-2.000	-2.000	-2.000	-2.000	76	109	862.9	4.677	50.92	0.30
8	C	-0.444	0.667	0.000	-2.000	2.000	-2.000	-2.000	2.000	76	117	661.7	4.781	40.47	0.38
9	C	-1.111	-0.333	-1.000	-2.000	0.000	-2.000	-2.000	1.000	76	104	834.3	6.026	39.02	0.23
10	C	-0.722	0.333	-0.500	-2.000	1.000	-2.000	-2.000	2.000	82	115	1098.0	4.706	40.95	0.32
11	C	-1.222	-0.667	-1.000	-2.000	0.000	-2.000	-2.000	0.000	71	104	934.3	5.517	47.97	0.24
12	C	-1.278	-0.333	-1.500	-2.000	-1.000	-2.000	-2.000	2.000	80	116	700.5	4.800	28.57	0.33
13	C	-1.111	-1.333	-0.500	-1.500	0.000	-1.000	-2.000	-2.000	79	108	882.9	4.586	34.90	0.22
14	C	-1.389	-0.667	-1.500	-2.000	-1.000	-2.000	-2.000	1.000	81	111	722.0	4.279	32.95	0.21
15	C	0.611	0.333	1.000	0.500	0.000	2.000	-1.000	2.000	89	105	565.4	3.479	21.89	0.10
16	C	-1.444	-1.333	-1.000	-2.000	0.000	-2.000	-2.000	-2.000	75	103	1126.0	6.232	43.50	0.24
17	C	-1.278	-0.333	-1.500	-2.000	-1.000	-2.000	-2.000	2.000	84	102	673.5	4.304	28.64	0.18
18	C	-0.722	0.333	-0.500	-2.000	1.000	-2.000	-2.000	2.000	85	109	654.6	4.572	27.80	0.19
19	C	-1.556	-0.667	-2.000	-2.000	-2.000	-2.000	-2.000	2.000	78	98	612.1	5.694	28.52	0.09
20	CPP	-1.778	-1.333	-2.000	-2.000	-2.000	-2.000	-2.000	0.000	94	115	497.9	2.764	21.62	0.10
21	CPP	-1.444	-1.333	-1.500	-1.500	-2.000	-1.000	-2.000	0.000	90	115	417.6	2.368	22.46	0.15
22	CPP	-1.500	-1.000	-1.500	-2.000	-1.000	-2.000	-2.000	0.000	58	97	1297.2	6.187	76.89	0.36
23	CPP	-1.056	-0.667	-1.000	-1.500	-1.000	-1.000	-2.000	1.000	93	115	361.9	2.848	16.99	0.12
24	CPP	-1.000	-1.000	-1.000	-1.000	-2.000	0.000	-2.000	1.000	77	101	912.5	2.371	34.06	0.18
25	CPP	-1.000	0.000	-1.000	-2.000	0.000	-2.000	-2.000	2.000	80	101	636.9	3.544	30.66	0.09
26	CPP	-0.944	-0.333	-0.500	-2.000	1.000	-2.000	-2.000	0.000	85	115	1017.4	5.965	25.17	0.19
27	CPP	0.000	0.000	0.500	-0.500	0.000	1.000	-2.000	2.000	97	119	463.1	3.023	20.36	0.18
28	CPP	-1.889	-1.667	-2.000	-2.000	-2.000	-2.000	-2.000	-1.000	74	106	1507.5	5.365	52.64	0.31
29	CPP	-1.889	-1.667	-2.000	-2.000	-2.000	-2.000	-2.000	-1.000	81	113	787.4	4.492	32.88	0.21
30	CPP	-0.444	-0.333	0.000	-1.000	0.000	0.000	-2.000	1.000	87	115	435.2	2.894	22.66	0.16
31	Java	-0.111	0.667	0.500	-1.500	2.000	-1.000	-2.000	2.000	105	140	204.5	1.948	10.04	0.26
32	Java	-1.444	-1.333	-1.500	-1.500	-2.000	-1.000	-2.000	0.000	108	149	155.3	1.288	8.65	0.33
33	Java	0.000	0.000	0.500	-0.500	0.000	1.000	-2.000	2.000	102	124	304.2	2.178	11.27	0.11
34	Java	0.556	0.667	1.000	0.000	1.000	1.000	-1.000	2.000	112	150	133.4	1.444	7.74	0.27
35	Java	0.833	1.000	1.500	0.000	2.000	1.000	-1.000	2.000	112	147	151.0	1.573	7.38	0.22
36	Java	-1.222	-0.667	-1.500	-1.500	-2.000	-1.000	-2.000	2.000	81	112	713.6	3.405	29.56	0.17
37	Java	0.167	0.000	0.500	0.000	0.000	1.000	-1.000	1.000	103	142	217.8	1.837	10.80	0.31
38	Java	-0.500	0.000	-0.500	-1.000	0.000	-1.000	-1.000	1.000	110	124	197.8	1.860	7.37	0.03
39	Java	-1.167	-1.000	-1.000	-1.500	-1.000	-1.000	-2.000	0.000	96	129	274.8	2.255	16.22	0.24
40	Java	-1.389	-0.667	-1.500	-2.000	-1.000	-2.000	-2.000	1.000	94	109	395.2	3.363	15.17	0.05
41	Java	-0.333	0.000	0.000	-1.000	0.000	0.000	-2.000	2.000	105	145	185.6	1.423	10.81	0.31
42	Java	-0.611	-0.333	-0.500	-1.000	-1.000	0.000	-2.000	2.000	82	110	658.7	3.227	27.61	0.17
43	Java	-0.778	-0.333	-0.500	-1.500	0.000	-1.000	-2.000	1.000	105	138	189.3	1.727	10.77	0.21
44	Java	-0.667	0.000	-0.500	-1.500	0.000	-1.000	-2.000	2.000	102	138	217.2	1.922	11.23	0.27
45	Java	-0.111	0.667	0.000	-1.000	1.000	-1.000	-1.000	2.000	99	141	243.0	2.225	14.60	0.38
46	Java	-0.778	-0.333	-1.000	-1.000	-1.000	-1.000	-1.000	1.000	101	132	250.8	2.096	11.93	0.16
47	Java	0.278	0.333	0.500	0.000	0.000	1.000	-1.000	2.000	113	141	147.9	1.783	7.12	0.37
48	Java	-0.889	-0.667	-0.500	-1.500	0.000	-1.000	-2.000	0.000	92	132	248.6	2.379	19.20	0.38
49	Java	-0.500	0.000	0.000	-1.500	1.000	-1.000	-2.000	1.000	98	125	362.7	2.768	12.88	0.17
50	Java	-0.722	0.333	-0.500	-2.000	1.000	-2.000	-2.000	2.000	88	125	349.9	2.869	19.92	0.35
51	Java	-0.111	0.667	0.500	-1.500	2.000	-1.000	-2.000	2.000	101	139	221.7	1.800	12.58	0.28
52	Java	-0.778	-0.333	-0.500	-1.500	0.000	-1.000	-2.000	1.000	99	136	233.2	2.393	13.48	0.34
53	Java	-1.167	-1.000	-1.000	-1.500	-1.000	-1.000	-2.000	0.000	100	138	229.4	1.928	12.97	0.30
54	Java	0.278	0.333	0.500	0.000	0.000	1.000	-1.000	2.000	97	135	259.9	2.294	14.63	0.34
55	Java	-0.389	0.333	-0.500	-1.000	0.000	-1.000	-1.000	2.000	96	135	274.8	2.261	15.75	0.35
56	Java	0.889	0.667	1.000	1.000	0.000	2.000	0.000	2.000	104	143	190.6	1.756	11.34	0.38

57	Java	-0.722	0.333	-0.500	-2.000	1.000	-2.000	-2.000	2.000	104	145	186.2	1.841	11.48	0.39
58	Java	0.889	0.667	1.000	1.000	0.000	2.000	0.000	2.000	100	139	209.9	1.724	14.20	0.37
59	Java	-1.778	-1.333	-2.000	-2.000	-2.000	-2.000	-2.000	0.000	93	129	347.8	2.342	15.29	0.26
60	Java	-1.889	-1.667	-2.000	-2.000	-2.000	-2.000	-2.000	-1.000	93	131	253.3	1.774	15.64	0.31
61	Java	-0.111	-0.333	0.000	0.000	-1.000	1.000	-1.000	1.000	97	137	280.2	2.103	14.55	0.35
62	Java	0.000	0.000	0.000	0.000	-1.000	1.000	-1.000	2.000	106	148	167.0	1.609	10.69	0.38
63	Java	0.611	0.333	1.000	0.500	0.000	2.000	-1.000	2.000	99	139	207.1	1.503	12.92	0.36
64	Java	-1.167	-1.000	-1.000	-1.500	-1.000	-1.000	-2.000	0.000	95	133	338.0	2.467	15.36	0.30
65	Java	-1.167	-1.000	-1.000	-1.500	-1.000	-1.000	-2.000	0.000	102	132	239.0	1.842	10.18	0.17
66	Java	-1.278	-0.333	-1.500	-2.000	-1.000	-2.000	-2.000	2.000	93	113	568.8	3.086	15.55	0.06
67	Java	-0.944	-0.333	-1.000	-1.500	-1.000	-1.000	-2.000	2.000	93	130	329.3	2.776	18.29	0.29
68	Java	-0.500	0.000	-0.500	-1.000	0.000	-1.000	-1.000	1.000	105	141	219.7	1.819	10.29	0.22
69	Java	-1.167	-1.000	-1.000	-1.500	-1.000	-1.000	-2.000	0.000	96	128	309.4	2.232	15.52	0.21
70	Java	-0.500	0.000	-0.500	-1.000	0.000	-1.000	-1.000	1.000	99	133	229.1	1.895	14.05	0.28
71	Java	0.000	0.000	0.000	0.000	-1.000	1.000	-1.000	2.000	100	144	187.1	1.757	13.51	0.42
72	Java	-1.222	-0.667	-1.500	-1.500	-2.000	-1.000	-2.000	2.000	111	145	133.8	1.757	8.40	0.20
73	Java	-1.056	-0.667	-1.000	-1.500	-1.000	-1.000	-2.000	1.000	89	121	473.5	2.974	20.13	0.20

Table 22. SMM and MI results